Verification of Graph Transformation Systems with k-Inductive Invariants

Johannes Dyck

Dissertation

zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften
– doctor rerum naturalium (Dr. rer. nat.) –

eingereicht an der Digital-Engineering-Fakultät der Universität Potsdam

Systemanalyse und Modellierung
Hasso-Plattner-Institut für Digital Engineering
Potsdam, 18. Dezember 2019

Betreuer: Prof. Dr. Holger Giese

Gutachter und Gutachterinnen:
Prof. Dr. Holger Giese
Prof. Dr. Arend Rensink
Prof. Dr. Heike Wehrheim

Published online at the
Institutional Repository of the University of Potsdam:
https://doi.org/10.25932/publishup-44274
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-442742

Acknowledgment

I am deeply grateful to Professor Holger Giese for giving me the opportunity to pursue a doctorate in the exciting field of modeling, system analysis, and, more specifically, formal verification of graph transformation systems. I would like to thank him for his supervision, feedback, and support. I am also grateful to Professor Arend Rensink and Professor Heike Wehrheim for taking the time to review my thesis. Furthermore, I would like to express my gratitude to Professor Felix Naumann, who agreed to chair my doctoral examination board, and Professor Tobias Friedrich and Professor Mathias Weske, who agreed to participate.

My work in the last years would not have been the same without my present and former colleagues at the System Analysis and Modeling group. I feel privileged to be a part of this group and to have been able to work alongside colleagues, former colleagues, and friends in a friendly, open, and productive environment. I am particularly grateful to Dr. Basil Becker and Dr. Leen Lambers, who first introduced me to the wonderful world of graph transformation systems and supported me over the years. I would like to thank Thomas Brand for sharing an office for several years and for a lot of politically charged lunch breaks. My thanks also go to Kerstin Miers for her support and encouragement.

Finally, my thanks go to my family and friends inside and outside the institute. I would not be where and who I am today without their continued support and friendship.

Abstract

With rising complexity of today's software and hardware systems and the hypothesized increase in autonomous, intelligent, and self-* systems, developing correct systems remains an important challenge. Testing, although an important part of the development and maintainance process, cannot usually establish the definite correctness of a software or hardware system – especially when systems have arbitrarily large or infinite state spaces or an infinite number of initial states. This is where formal verification comes in: given a representation of the system in question in a formal framework, verification approaches and tools can be used to establish the system's adherence to its similarly formalized specification, and to complement testing.

One such formal framework is the field of graphs and graph transformation systems. Both are powerful formalisms with well-established foundations and ongoing research that can be used to describe complex hardware or software systems with varying degrees of abstraction. Since their inception in the 1970s, graph transformation systems have continuously evolved; related research spans extensions of expressive power, graph algorithms, and their implementation, application scenarios, or verification approaches, to name just a few topics.

This thesis focuses on a verification approach for graph transformation systems called k-inductive invariant checking, which is an extension of previous work on 1-inductive invariant checking. Instead of exhaustively computing a system's state space, which is a common approach in model checking, 1-inductive invariant checking symbolically analyzes graph transformation rules – i.e. system behavior – in order to draw conclusions with respect to the validity of graph constraints in the system's state space. The approach is based on an inductive argument: if a system's initial state satisfies a graph constraint and if all rules preserve that constraint's validity, we can conclude the constraint's validity in the system's entire state space – without having to compute it.

However, inductive invariant checking also comes with a specific drawback: the locality of graph transformation rules leads to a lack of context information during the symbolic analysis of potential rule applications. This thesis argues that this lack of context can be partly addressed by using k-induction instead of 1-induction. A k-inductive invariant is a graph constraint whose validity in a path of $k-1$ rule applications implies its validity after any subsequent rule application – as opposed to a 1-inductive invariant where only one rule application is taken into account. Considering a path of transformations then accumulates more context of the graph rules' applications.

As such, this thesis extends existing research and implementation on 1-inductive invariant checking for graph transformation systems to k-induction. In addition, it proposes a technique to perform the base case of the inductive argument in a symbolic fashion, which allows verification of systems with an infinite set of initial states. Both k-inductive invariant checking and its base case are described in formal terms. Based on that, this thesis formulates theorems and constructions to apply this general verification approach for typed graph transformation systems and nested graph constraints – and to formally prove the approach's correctness.

Since unrestricted graph constraints may lead to non-termination or impracticably high execution times given a hypothetical implementation, this thesis also presents a restricted verification approach, which limits the form of graph transformation systems and graph constraints. It is formalized, proven correct, and its procedures terminate by construction. This restricted approach has been implemented in an automated tool and has been evaluated with respect to its applicability to test cases, its performance, and its degree of completeness.

Zusammenfassung

Durch die Komplexität heutiger Software- und Hardwaresysteme und den vermuteten Anstieg der Zahl autonomer und intelligenter Systeme bleibt die Entwicklung korrekter Systeme eine wichtige Herausforderung. Obwohl Testen ein wichtiger Teil des Entwicklungszyklusses ist und bleibt, reichen Tests üblicherweise nicht aus, um die Korrektkeit eines Systems sicherzustellen – insbsondere wenn Systeme beliebig große oder unendliche Zustandsräume oder unendlich viele mögliche initiale Zustände aufweisen. Formale Verifikation nimmt sich dieses Problems an: Nach Darstellung des Systems in einem formalen Modell können Verifikationsansätze und Werkzeuge angewendet werden, um zu analysieren, ob das System seine Spezifikation erfüllt.

Ein verbreiteter Formalismus für derartige Modelle sind Graphen und Graphtransformationssysteme. Diese Konzepte basieren auf etablierten mathematischen Grundlagen und sind ausdrucksstark genug, um komplexe Software- oder Hardwaresysteme auf verschiedenen Abstraktionsstufen zu beschreiben. Seit ihrer Einführung in den 70er-Jahren wurden Graphtransformationssysteme stetig weiterentwickelt; entsprechende Forschung thematisiert beispielsweise Ausdrucksstärke, Graphalgorithmen, Anwendungsbeispiele oder Verifikationsansätze.

Diese Arbeit beschäftigt sich mit der Verifikation k-induktiver Invarianten für Graphtransformationssysteme – einem Ansatz, der eine existierende Technik zur Verifikation 1-induktiver Invarianten erweitert. Anstatt den Zustandsraum eines Systems zu berechnen, überprüft Verifikation mit 1-Induktion Verhalten (Graphtransformationsregeln) symbolisch, um Schlussfolgerungen zur Gültigkeit von Graphbedingungen zu ziehen. Die Idee basiert auf dem Prinzip eines Induktionsbeweises: Falls der initiale Zustand eines Systems eine Bedingung erfüllt und falls alle Regeln die Erfüllung der Bedingung bewahren, kann auf die Gültigkeit der Bedingung im gesamten Zustandsraum geschlossen werden, ohne diesen tatsächlich zu berechnen.

Allerdings bringt dieser Ansatz auch spezifische Nachteile mit sich: Die lokale Natur der Anwendung von Graphregeln führt zu einem Mangel an Kontext während der symbolischen Analyse möglicher Regelanwendungen. Diese Arbeit führt aus, dass dieser Mangel an Kontext teilweise behoben werden kann, indem k-Induktion statt 1-Induktion verwendet wird. Eine k-induktive Invariante ist eine Graphbedingung, deren Gültigkeit in einem Pfad von $k-1$ Regelanwendungen die Gültigkeit nach jeder etwaigen weiteren Regelanwendung zur Folge hat. Durch die Berücksichtigung solcher Pfade von Transformationen steht mehr Kontext während der Analyse zur Verfügung als bei der Analyse nur einer Regelanwendung bei 1-Induktion.

Daher erweitert diese Arbeit bestehende Forschungsergebnisse und eine Implementierung zur Verifikation 1-induktiver Invarianten um k-Induktion. Zusätzlich wird eine Technik vorgestellt, die auch die Analyse der Induktionsbasis symbolisch ausführt. Dies erlaubt die Verifikation von Systemen mit einer unendlichen Zahl an möglichen initialen Zuständen. Sowohl k-induktive Invarianten als auch deren Induktionsbasis werden – für Graphtransformationssysteme – formal beschrieben. Basierend darauf stellt diese Arbeit Theoreme und Kontruktionen vor, die diesen Verifikationsansatz mathemathisch umsetzen und seine Korrektheit beweisen.

Da jedoch uneingeschränkte Graphbedingungen in einer möglichen Implementierung zu Nichtterminierung oder langen Ausführungszeiten führen, stellt diese Arbeit auch einen eingeschränkten Verifikationsansatz vor, der die Form der zugelassenen Graphtransformationssysteme und Graphbedingungen in Spezifikationen einschränkt. Auch dieser Ansatz wird formalisiert, bewiesen – und das Verfahren terminiert per Konstruktion. Der Ansatz wurde in Form eines automatisch ausführbaren Verifikationswerkzeugs implementiert und wurde in Bezug auf seine Anwendbarkeit, Performanz und des Grades der Vollständigkeit evaluiert.

Contents

1. Introduction

Given the ever rising complexity of modern software and hardware systems, developing and maintaining correct software remains an important challenge in software development and, on a more conceptual level, computer science. This applies to a variety of different types of systems: cyber-physical systems, which often overlap with safety-critical systems, automated processes in model-driven engineering, such as model transformations, or protocol specifications, to name just a few.

While testing is and will remain important for a variety of types of software and hardware systems, it cannot usually guarantee that a system in question is free of errors. Formal verification, on the other hand, can establish these guarantees, but faces other challenges: the necessity of a formal framework systems have to conform to or be mapped to, the undecidability of important problems, and the computational complexity of automated verification approaches.

We consider the following elements as essential parts of systems and system specifications:

System metamodel: describes the entities and associations that may appear in system states.

System states describe situations that may occur in the system. A system has an initial state and, during execution, a current state describing its current situation, which may change by execution of system behavior.

System behavior describes change between system states.

Properties can be satisfied or violated by system states.

System: consists of a metamodel, an initial state, and a specification of system behavior.

System state space: consists of all system states that are reachable from the system's initial state by execution of system behavior.

Here, the term metamodel refers to a means of describing entities appearing in system states. Examples include UML class diagrams and type graphs. Then, all system states are instances of – or typed over – the respective system's metamodel. Properties in the sense used here mean state properties – as opposed to, for example, temporal properties specified in computational tree logic or linear temporal logic. State properties appear in a number of types in this thesis, most importantly as safety properties whose verification is the central problem we want to solve:

Verification Problem 1.1. *Given a system that is defined by a metamodel, an initial system state, and specification of system behavior and given a set of safety properties, does every state in the system's state space satisfy the safety properties?*

An important example and entire class of formal verification approaches is model checking [BK08]. Model checking approaches focus on analyzing a system's state space, which is the set of all states the system can assume, and establishing or disproving the validity of desirable properties with respect to individual states, sets of states, or traces of states. Model checking, like other formal verification approaches, requires a mathematical formalization of system states, system behavior, and properties that can be satisfied by the system, individual system states, or traces of states [BK08]. In general, model checkers explore the system's state space, either exhaustively or symbolically, such as by abstraction, then verify the specified properties and produce a corresponding result. As such, model checking can be applied to solve Verification Problem 1.1.

In order to apply a verification approach to specific systems, we have to use a *formal model* that provides formalizations for the system elements listed above. In the literature, a number of formalizations for system behavior have been suggested and tooling – such as model checkers – has been built around them. Of the formal frameworks employed in model checking and formal verification approaches in general, each has their own unique advantages and disadvantages, often depending on the problem domain and specific verification scenario at hand. This thesis and the approach discussed herein focus on graph transformation systems, graphs, and graph constraints as one well-established formal model [EEPT06, Roz97, EEKR99, HP09, EGH^{+}14] with existing and ongoing research.

Graphs are capable of representing complex structures consisting of nodes, which represent entities, and edges, which represent relationships and connections between entities. Intuitively, a graph describes the specific state of a system. Changes between states – in other words, system behavior – is then described by graph transformation rules in a graph transformation system. Graph constraints can be used to specify properties; their validity can then be verified in specific graphs.

Graphs and graph transformation systems have been the subject of research in computer science since the 1970s [EEPT06]. This research has spanned a large spectrum and multiple dimensions. Notions of graph transformation systems have been extended in complexity and expressive power [EH86, HHT96, HP09] and analyzed with respect to their relevance and applicability in practice [NSM03], and verification tools specifically tailored to graph transformation systems have been introduced [Tae00, Ren04, ABJ^{+}10].

One reason why graphs are well suited for formal verification of complex systems is their connection to modeling – of both software and specifications – and model-driven engineering [EEPT06]. Models typically contain a graph structure; for instance, a UML object diagram has objects as nodes and associations as edges. Similarly, a class diagram has classes as nodes and associations or inheritance relationships as edges. Objects (nodes of the object diagram) can be mapped to their respective classes (nodes in the class diagram); the same holds for edges. In the world of graphs, the class diagram would be a type graph with classes as node types and connections as edge types. Then, object diagrams would be typed graphs, which are typed over a type graph – the class diagram.

While models may well be used independently from each other throughout different stages of the software (or hardware) development process, model-driven engineering relies on interlinked models [Ken02] as artifacts of the development process. Often, models start out as a rather abstract view of the system to be developed and may then be gradually refined – sometimes with the goal to generate executable code from suitably refined models, other times with the aim of feeding models to an interpreter or engine capable of executing the models themselves. Refinement of models – and more generally, any changes to models – may be performed by model transformations [Ken02], often automatically. With models interpreted as graphs, model transformations can be specified by (and seen as) graph transformation rules and their execution as graph transformations. Then, tooling for formal verification of graph transformations can be applied, provided it supports graph transformations and their properties to the extent required by the scenario in question. Even without the context of a model-driven engineering process, model transformations have their use: we might want to collectively migrate a set of models to a newer version of the corresponding metamodel, convert a model specified in a domain-specific language (DSL) to a more general format, or present a system of communicating automata as a more illustrative sequence diagram while preserving its semantics [GL12].

As we will see later, the latter example – communicating automata, sequence diagrams, and semantics preservation – already hints at one of this thesis's examples – and, more importantly, at a key aspect to be considered when applying model transformations: correctness. This is particularly important for model transformations in the context of model-driven engineering, where many transformations, including code generation, may happen in an automated fashion. Depending on the number, size, and complexity of the models, model transformations, and notion of correctness, manual inspection may be infeasible, prompting the need for appropriate automated formal verification approaches.

Although it is a typical example, the use of graphs and graph transformations is not restricted to model transformations and applications in model-driven engineering. In theory, any system whose states conform to the basic structure of graphs – nodes and edges – can be represented by a graph. Where system behavior can be described by structural changes in those states, it can be modeled by graph transformation rules. In the literature, graphs and graph transformations have been used to represent systems from a range of different domains, and different static verification approaches have been suggested and applied. Examples include:

Car platooning system: cars (nodes) can create connections (edges) between each other to form platoons. The rules governing the creation, dissolution, and behavior of platoons are described by graph transformation rules, which are verified to guarantee certain properties [Pen09].

Leader election protocol: processes (nodes) will create, send, and receive messages (nodes) in order to determine a leader. Processes and messages are connected by edges; creating, sending, and handling messages is governed by graph transformation rules. Corresponding state spaces can be analyzed for liveness and safety [GdMR+12].

Java refactorings: graphs are used to represent the structure of a Java program (at design time). Elements like classes, methods, or fields are represented by nodes; their (containment) associations are represented by edges. Graph transformation rules describe refactoring rules, which can be verified for consistency preservation [BLD+11].

Model transformations: among many examples, graphs have been used to describe models of sequence diagrams and systems of communicating automata, with nodes representing lifelines, events, automata, states, transitions, and message, and edges representing their connections. Graph transformation rules are used to describe model transformations between both types of models and to describe model semantics. Formal verification can be applied to show that the model transformation in question is semantics-preserving [GL12].

Task scheduling system: tasks, a scheduler, and a processing unit are represented by nodes, with their connections described by edges. Graph transformation rules describe how tasks are created, scheduled for execution, and executed. The goal of the verification is to show that a processing unit executes at most two tasks at any given time [Ste15].

Whether or not graphs and graph transformations can be used to describe system states and system behavior is often a question of the system's complexity and whether is is matched by the expressive power of the underlying formalism (of graphs). Fortunately, since the inception of graphs and graph transformations as a formal notion, many extensions in the direction of expressive power have been suggested, formalized, and discussed, such as:

Application conditions: a generalization of (negative) application conditions [EH86, HHT96], application conditions allow, intuitively speaking, the specification of boolean conditions over the existence, absence, and connections of graphs and graph elements [HP09, EGH+14] and can be used to impose fine-grained restrictions on the applicability of graph transformation rules.

Graph constraints: a special type of application conditions applicable to (and satisfiable by) graphs [HP09]. They can, for example, be used to describe desirable or undesirable system properties.

HR* conditions: an extension of application conditions beyond the boundaries of first-order logic [Rad13].

Control conditions, which allow for fine-grained control of execution of system behavior by describing control flow – e.g. order and logical sequence – of rule applications [GdMR+12, Pen09].

Attributed graphs, which extend the purely structural concept of a graph by attributes of different types and values [EPT04, EEPT06, OL10b].

Timing: clocks can be seen as attributes of a distinguished type, allowing a more realistic specification of, for instance, cyber-physical systems [BG08b, HSE11, MGK17].

Probability, where non-deterministic choices between the application of applicable graph rules are made according to specified probabilities [KG12, MGK17].

However, having a formalism with the expressive power to properly describe a system is just the first step towards formal verification: the implementation of the approaches and tools for formal verification have to support the formalism to the extent required as well. This may result in possibly conflicting requirements: using a highly expressive formal model in automated verification generally increases the likelihood of undecidable verification problems and high computational effort. Formal models of lower expressive power may not be sufficient to accurately reflect the system and its properties.

1.1. Motivation and Characteristics of Symbolic Verification Approaches

Regardless of the expressive power of graph transformation systems, application conditions, and other extensions, some verification approaches may not be applicable to certain scenarios – or may be applicable, but ill suited – for two reasons: infinite state spaces and changing system parameters. To understand the first problem, consider model checking: one possible approach is to start at a system's initial state (graph), then consider all possible executable behavior (graph transformation rules) to compute all subsequent states, and repeat that process until all reachable system states have been found. As long as the number of system states is finite, systems can be explored and analyzed in this fashion. However, systems with an infinite number of possible states – i.e. an infinite state space – cannot be verified by this approach. And even for finite state spaces, the exponential complexity of the state space, often referred to as the state space explosion problem [BK08], may render explicit-state model checking [CHVB18, Ren08] approaches infeasible.

Given the approach outlined above, verification results are specific to the combination of an initial system state and the specification of system behavior – which in our case are a start graph and graph transformation rules. Changing the system's start graph or its behavior invalidates the previous verification result and requires executing the procedure again. With large state spaces and the resulting computational effort, iterative development of complex systems quickly becomes tedious and time-consuming. Furthermore, if we want to verify system behavior for an arbitrarily large (or infinite) set of possible initial states, the approach cannot be applied at all.

To better understand both problems, we will discuss a number of examples. The first example will also serve as this thesis's running example and illustrates the problem of changing system parameters.

Example 1.1 (shuttle protocol)**.** The core idea of our first example is very roughly based on the RailCab project[1], which has been used as an example in research before [BBG+06, BG08b, SW11]. Intuitively, a shuttle moves in different speed modes around a topology of connected

[1] `https://www.hni.uni-paderborn.de/en/business-computing-especially-cim/projects/railcab/`

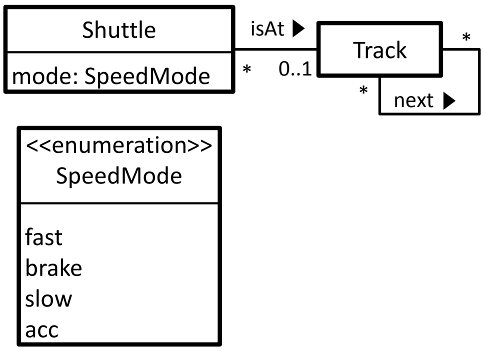


Figure 1.1. – UML class diagram for shuttle protocol example

tracks and, with respect to its speed modes, follows a certain protocol *[3]*. Figure 1.1 shows a UML class diagram of the system. We have:

tracks, which are unidirectionally connected by the next association,
shuttles, with each shuttle located on (at most) one track and having a
speed mode, which may be one of the four values fast, brake, slow, and acc(elerating).

There will be additional constraints restricting, for example, track topology, but those constraints cannot be represented as part of the class diagram and would have to be implemented as OCL constraints instead. We will reintroduce this example using type graphs and graph transformation systems after these concepts have been formally introduced in Chapter 2. However, as mentioned before, the type graph will strongly resemble the class diagram; typed graphs will then mimic object diagrams.

For a shuttle, switching between speed modes will always happen as part of shuttle movement. Intuitively, a shuttle can move from its current location to a subsequent (i.e. connected by next) track and, while doing so, may keep or change its speed mode as specified. In particular, shuttle behavior consists of the following actions, which will later be implemented as graph transformation rules:

s2s (slow2slow), where a shuttle in mode slow moves ahead without changing its speed mode,
f2f (fast2fast), where a shuttle in mode fast moves ahead without changing its speed mode,
s2a (slow2acc), where a shuttle in mode slow moves ahead and changes its speed mode to acc(elerating),
a2f (acc2fast), where a shuttle in mode acc moves ahead and changes its speed mode to fast,
f2b (fast2brake), where a shuttle in mode fast moves ahead and changes its speed mode to brake,
a2b (acc2brake), where a shuttle in mode acc moves ahead and changes its speed mode to brake, and
b2s (brake2slow), where a shuttle in mode brake moves ahead and changes its speed mode to slow.

Note that the protocol exhibits certain symmetries with the exception of a2b. There is no symmetric rule b2a: after starting to brake, the process needs to be completed before the shuttle may accelerate again.

If, in the track topology, two tracks have the same track as its successor, that latter track is a switch. We want to prevent shuttles from driving fast on a switch because, under certain circumstances, this may lead to the shuttle's derailment. In particular, we want to apply formal verification to prove the non-occurrence of this situation for our shuttle protocol.

Given an initial system state, we can apply behavioral rules to find a violation of the property sketched above. If behavioral rules involving high speed modes do not have some sort of safeguard, we will likely encounter the forbidden situation at some point, making our system unsafe. However, this insight is specific to said initial state and the specified behavior; changing one or both will lead to a different result. Thus, our aim cannot be to establish safety of the system with respect to specific initial states, which always include a specific track topology. Every change in track topology would require repeating the verification. Instead, we would like to prove safety of system behavior for all possible initial configurations. This is beyond the capabilities of the model checking approach outlined above. In particular, the infinite number of possible initial states requires establishing safety properties independent of initial configurations – in other words, we need a symbolic approach. Then, changes in track topology (offline) do not invalidate the verification result. Changes in system behavior, however, still do.

Note that this protocol is only an abstraction of an actual system with realistic shuttle movement. Here, timing and velocity as a continuous function of time is not considered. However, the non-discrete nature of changes in velocity are represented in an implicit fashion: a shuttle cannot switch its speed mode from slow to fast or vice versa. Instead, there are intermediate modes acc and brake. Thus, accelerating from slow to fast via acc requires at least two consecutive behavioral actions. Similarly, slowing down from fast via brake to slow takes time – in the sense of the execution of two behavioral rules. △

This example has illustrated the problem of changing initial configurations, up to the point of allowing an arbitrary large or infinite number of initial states. Our next example comes from the domain of model transformations and demonstrates the problem of infinite state spaces.

Example 1.2 (model transformation)**.** This example has been taken from existing work on the verification of behavior preservation for relational model transformations [*6*, *5*]. Type graphs and graph constraints are used to define source and target modeling languages for sequence charts and systems of communicating automata. A relational model transformation between instances of the modeling languages is specified by a triple graph grammar [Sch94, SK08], which consists of an axiom (i.e. initial state) and a number of (triple) graph transformation rules. Then, the approach aims to verify behavior preservation of this transformation by showing that all its instances – pairs of corresponding source and target models – are equivalent in their behavior. This requires establishing certain formal properties of the triple graph grammar's graph transformation rules and the semantics of source and target models.

The number of possible sequence charts and systems of communicating automata will be infinite. Hence, the model transformation – specified by a triple graph grammar – needs to be applicable to an infinite number of source and target models. A triple graph grammar, by execution of its transformation rules, simultaneously creates the source and target models and their correspondences (traceability information) alongside each other. Intuitively speaking, the grammar thusly creates all pairs of source and target models that can be transformed into each other, i.e. that are instances of the model transformation. Hence, the triple graph grammar's state space will be infinite – and any verification approach applicable to this particular problem needs to handle infinite state spaces.

There are significant differences to the shuttle protocol in Example 1.1 (p. 4). For the shuttle protocol, state spaces are finite for each start graph – unless, for example, system behavior includes the addition of tracks or shuttles. Conversely, the state space of a triple graph grammar specifying the example model transformation will be infinite. On the other hand, the triple graph grammar will only have one initial state – its axiom. For the shuttle protocol, changes in track topology (offline, i.e. while the system is not running) are to be expected at some point. △

Table 1.1. – Overview and comparison of examples

	Shuttle protocol (Example 1.1)	Model transformation (Example 1.2)	Model semantics (Example 1.3)
Initial system state	Track topology and shuttle location	Axiom of triple graph grammar	Specific source or target model
System behavior	Shuttle movement and speed modes	Model transformation rules	Model semantics
Cardinality of initial states	Infinite set	One	Infinite set
Cardinality of state space	Finite	Infinite	Finite

Example 1.3 (model semantics). The approach to verifying behavior preservation for model transformations sketched in Example 1.2 (p. 6) requires establishing certain properties of the model transformation and of source and target model semantics. Model semantics are specified by graph transformation rules that can then be applied to source and target models. Then, each source (or target) is the initial state of its own state space created by application of source (or target) semantics. In order to verify the semantic properties required, we need to investigate all source and target models – initial states – and their state spaces. With an infinite number of such initial states, we have another example where explicit state space exploration is impossible.

Again, there are differences to the previous examples, which are illustrated in Table 1.1. In Example 1.2 (p. 6), there is only one initial state – the axiom of the triple graph grammar. In Example 1.1 (p. 4), we have a set of initial states – different track topologies – but successful verification for only one initial state or a finite set of initial states may be useful in certain cases. In Example 1.3, the set of initial states is infinite. Results that consider only a finite subset are usually insufficient to derive meaningful properties of the model transformation with respect to behavior preservation.

With respect to the cardinality of state spaces, we have a finite state space (per initial graph) in Example 1.1 (p. 4) and an infinite state space in Example 1.2 (p. 6). In this example, due to restrictions on source and target models, we have finite state spaces per initial graph, too. △

We could further distinguish between the existence and absence of cycles in state spaces. In a cyclic state space, we can have infinite traces of behavioral steps in a finite state space. However, analyzing infinite state spaces is harder than analyzing finite and cyclic state spaces with possible infinite traces – at least when state properties are concerned. Hence, we do not require this distinction here.

These examples illustrate the need for symbolic and static verification approaches capable of handling infinite state spaces and systems with an arbitrary or infinite number of initial states. However, with all their advantages, symbolic and static verification approaches also face problem-inherent challenges that are related to three important properties: *termination*, *soundness*, and *completeness*.

Definition 1.4 (termination). Termination *refers to the assurance that a verification approach will always terminate and yield a result for any system.*

Definition 1.5 (soundness). *We say that a verification approach is* sound *with respect to the verification of safety properties, if every unsafe system is correctly classified as unsafe.*

Definition 1.6 (completeness)**.** *We say that a verification approach is* complete *with respect to the verification of safety properties, if every safe system is correctly classified as safe.*

These three properties are linked and, in many cases, cannot all be guaranteed at the same time. For example, if the underlying problem or formalism is undecidable in general, an approach is necessarily unsound, incomplete, or not terminating. With respect to the verification of safety properties, especially in the domain of safety-critical systems, soundness is often more important than completeness: If an unsafe system is classified as safe (unsound approach) and, as a result, it is allowed to operate, it may have disastrous consequences. If a safe system is classified as unsafe (incomplete approach) and shut down as a result, this is more inconvenient than catastrophic. The importance of termination depends on its manifestation and the specific case: if the approach does not yield any result, this is problematic; if the approach yields partial results, then continues execution indefinitely, these results may already be sufficient. On the other hand, termination may sometimes not be enough: if an approach is guaranteed to terminate, but requires an unreasonably high amount of computation time (say, a year), its application is infeasible for most practical purposes.

We can also define soundness and completeness by analyzing verification results. With respect to the verification of safety properties in the context of this thesis, systems are either classified as safe or declared unsafe by virtue of a number of counterexamples. Then, results fall into four categories: *true positives*, *true negatives*, *false positives*, and *false negatives*:

Definition 1.7 (categories of results)**.** *A* true positive *is a result that classifies a safe system as safe.*

A true negative *is a counterexample for an unsafe system that can occur during system behavior.*

A false positive *is a result that classifies an unsafe system as safe.*

A false negative *is a counterexample for an unsafe system that cannot occur during system behavior.*

Of those cases, the first two are, by definition, valid results – a sound and complete approach will only yield true positives and true negatives. An unsound approach will have false positives, which are the most dangerous of the four categories, for reasons explained above. An incomplete approach will have false negatives, also called *spurious counterexamples.* The risk of unsound, incomplete, or non-terminating approaches may be traced back to two main problem-inherent challenges of symbolic verification approaches:

Lack of context information. State spaces are not explicitly computed, but treated symbolically. This process abstracts from some information otherwise available in individual states. Also, reachability cannot be taken into account outside a small local context. As a result, symbolic approaches can be incomplete or unsound.

Computational effort. Since we need to handle infinite systems, certain subproblems may be undecidable – or, where implementation and execution is concerned, infeasible to solve. As a result, approaches may not guarantee termination – and even with termination assured, approaches may require unreasonably high computational effort.

The problem of high computational effort is also connected to the degree of expressive power supported by the approach: the more expressive the formalism, the higher the risk a problem is undecidable or requires infeasibly expensive computations. There is also a connection between the size of certain system elements and computational effort. Hence, support for a high degree of expressive power and complexity is not always desirable – instead, a balance between computational effort and expressiveness may be preferable.

There is a number of approaches (with implementations) capable of performing static and symbolic analysis of graph transformation systems and graph grammars with respect to verifying the validity of state properties in state spaces. They will be considered in more detail in Chapter 8. All such approaches need to find a balance between allowing expressive specifications and and termination, soundness, completeness, and performance. Some approaches or their implementations consider only one initial state or one error graph; most have limitations with respect to the specification of behavior and safety properties [KK08, Stü16, Ste15]. More expressive approaches [Pen09, Pos13], on the other hand, may struggle with performance because their general nature makes optimizations for specific problems challenging.

In the following, related approaches are listed in roughly chronological order:

– Between 2006 and 2009, Pennemann, Habel, and Rensink [HPR06, Pen08a, Pen08b, HP09, Pen09] have described an approach for the verification of graph programs with respect to their postconditions and preconditions. The approach is implemented in a tool named Enforce, which relies on a satisfiability solver SeekSat and theorem prover ProCon. Both the formalization and implementation support a class of properties equivalent to first-order logic. However, that level of expressive power may lead to high execution times or high amounts of required memory in certain cases *[1]*.

– Between 2010 and 2014, Poskitt and Plump [PP10, PP12, PP13, Pos13, PP14] have also addressed verification of graph programs with respect to postconditions and preconditions. In contrast to the approach by Pennemann and Habel, the approach considers more expressive conditions. In addition, verification is meant to be partly based on human interaction, not fully automated execution.

– Between 2011 and 2015, Steenken, Wonisch, and Wehrheim [SW11, SWW11, Ste15] introduced an approach and implementation addressing verification of graph grammars by shape abstraction, which is applicable to infinite-state systems. While not necessarily a restriction of the underlying approach, the implementation requires singular initial states and does not support negative application conditions.

– Between 2012 and 2017, Stückrath and König [KS12, KS14, Stü16, KS17] have proposed a static verification approach based on the idea of well-structured graph transformation systems: if the systems fulfill certain properties, reachability of an error state from an initial state (singular) can be checked for systems with an infinite state space.

– More recently, in 2018, Rabbi, Kristensen, and Lamo [RKL18] have described a static verification approach that verifies conformance preservation of graph transformation rules with a focus on model transformations and well-formedness constraints of metamodels; an implementation is intended for the future.

Outside the world of symbolic approaches, there are well-known (explicit) model checkers GROOVE[2] [Ren04, Ren08, GdMR+12] and Henshin[3] [ABJ+10], both of which address graph transformation systems and graph constraints.

1.2. 1-Inductive Invariant Checking and Open Problems

This thesis focuses on the verification approach of inductive invariant checking for graph transformation systems and its extension. Inductive invariant checking is closely related to the principles of mathematical and structural induction. The basic structure of an inductive argument consists of

[2] http://groove.cs.utwente.nl/

[3] https://www.eclipse.org/henshin/

1. a base case, where a property is shown for one element or structure and
2. an inductive step, where it is shown that validity of the property for one element or structure implies its validity in a derived element or structure.

Then, we can argue that the property's validity in the base case and its extendability via the inductive step implies the property's validity in all elements subsequentially derived from the base case. Using an inductive argument requires some sort of ordering between elements or a specification how structures are derived from each other – otherwise, is is unclear how the inductive step should reason from one element to the property's validity in another. For example, induction can be performed over the set of natural numbers: the inductive step reasons about the implications of a property's validity in one number for the property's validity in that number's successor.

In the following, we will remain on the abstract level of systems and system state spaces as in Verification Problem 1.1 (p. 1). A formalization with respect to graphs and graph transformation systems will appear after we have introduced the formal foundations. That leaves us with the following generic definition of an inductive invariant:

Definition 1.8 (inductive invariant)**.** *An* inductive invariant *is a property whose validity in a system state implies its validity in the next step after a single execution step of system behavior.*

By this definition, an inductive invariant is a property whose validity is preserved by system behavior. This works as a contraposition, too: an inductive invariant is a property whose violation after a behavioral action implies violation of the property before execution of the action. Similar to before, we require a notion of how elements – states – are derived from each other. Here, this happens by considering how states change by execution of system behavior. As a result, an inductive invariant is specific to the system's behavior, but not to the base case of an inductive argument. Since the definition of inductive invariants only considers a single system state and a single step of system behavior, we also refer to this type of inductive invariant as a 1-inductive invariant – and to a possible verification approach as 1-inductive invariant checking.

A 1-inductive invariant represents the inductive step of an inductive argument. In particular, with respect to our generic notions of systems, state spaces, and system behavior, an inductive argument consists of:

1. the base case, where a property is shown for a system's initial state and
2. the inductive step, where it is shown that the property is an inductive invariant.

In other words, if an inductive invariant also holds in a system's initial state (base case), it will hold in all states of the system's state space. Then, the property is an *operational invariant*:

Definition 1.9 (operational invariant)**.** *An* operational invariant *of a system is a property that is valid in the system's entire state space.*

A formal approach and tooling for 1-inductive invariant checking for graph transformation systems has been introduced [BBG+06], applied [BG08b, BLD+11, GL12], and extended [Dyc12] in earlier work. Relevant publications are listed below in chronological order; Table 1.2 shows them in a more compact fashion.

– In 2006, Becker et al. [BBG+06] introduced the original approach and its application in the mechatronic domain, with examples based on the RailCab project of autonomously

Table 1.2. – Overview of previous work

Authors (year)	Extensions of formal model	Application examples
Becker et al., 2006 [BBG+06]	Limited form of application conditions, rules with priorities	RailCab
Becker and Giese, 2008 [BG08b]	Continuous attributes, clocks, urgent rules	RailCab
Becker and Giese, 2008 [BG08a]	–	RailCab
Becker et al., 2011 [BLD+11]	–	Java refactorings
Dyck, 2012 [Dyc12]	Extended form of application conditions, no support for attributes	–
Giese and Lambers, 2012 [GL12]	–	Behavior preservation of model transformations

driving shuttles. The approach's formal model supported a limited form of graph constraints and negative application conditions. Application of transformation rules could be steered with rule priorities. The original implementation also supported a symbolic algorithm that encoded graphs and graph constraints using binary decision diagrams. Back then, this algorithm was more efficient than using the graph data structures directly.

- In 2008, Becker and Giese [BG08b, BG08a] described an extended formal model with continuous attributes, clocks, and timing. Transformation rules could be denoted as urgent; urgent rules require immediate application as soon as the rule is applicable.
- In 2011, Becker et al. [BLD+11] applied the approach in a multi-layered scheme to verify consistency preservation of Java refactorings.
- In 2012, the author of this thesis [Dyc12] extended the formal model and implementation towards support for more complex negative application conditions in rules and graph constraints. Since there are still restrictions in expressiveness, the model is referred to as the *restricted formal model*; furthermore, attributes are not supported in the extension.
- Also in 2012, Giese and Lambers [GL12] applied the approach in a two-layered scheme to verify behavior preservation of model transformations.

Intuitively, the basic idea of the approach is as follows: symbolic encodings for the violation of a property and the result of the application of a graph transformation rule are combined to form a potential counterexample. Then, the rule is applied in reverse direction to find the symbolic representation of the system state before rule application. If the property was already violated there, this sequence of one step is not a counterexample; otherwise, the property is not an invariant. While there usually is an infinite amount of these situations, their symbolic representation allows us to analyze them in a finite fashion.

Note that the verification of 1-inductive invariants as described above is indeed independent of specific state spaces and initial states. The validity of inductive invariants in a system's entire state space, however, is not: we need to verify the property's validity in the initial state, too. Hence, inductive invariant checking is a solution for the problem of infinite state spaces, but it addresses the question of an arbitrarily large number of initial states only partially. From a perspective of computational effort, it is very inexpensive to verify a property in one specific initial state – much less expensive than verifying the property in the entire state space. This makes the approach preferable to model checking for certain scenarions with a finite, but large amount of initial states. It is even applicable if there is a potentially infinite number of initial states – such as track topologies in Example 1.1 (p. 4) – if it is feasible to repeatedly verify changed initial states whenever necessary. However, analyzing infinitely many initial states in a finite fashion is not yet supported by this approach [BBG+06, Dyc12].

The decoupling of base case and inductive step can have adverse effects, too. In particular, it can lead to a lack of context information during the verification procedure as explained earlier. For example, the initial states – together with system behavior – directly influence the state space: elements not modified by any rule will remain the same in each state of the state space. If all allowed initial states have common properties with respect to unchanged elements, this information could be used during the inductive step. However, the approach and its verification of 1-inductive invariants cannot take this into account.

Regarding the three properties of verification approaches outlined in Definitions 1.4-1.6 (p. 7), we should note that the approach is sound and terminates [Dyc12]. Termination and reasonable performance is ensured by limiting the degree of expressive power – and by using symbolic representation. However, the approach is not complete: verification may result in false negatives.

Example 1.10. Consider the shuttle protocol from Example 1.1 (p. 4). We want to establish the safety property specifying the absence of a shuttle on a switch in speed mode fast. We need to show:

Base case: in the initial state, there is no fast shuttle located on a switch.

Inductive step: no rule application results in a fast shuttle positioned on a switch unless that situation was already present.

If both statements can be shown – i.e. if the safety property is a 1-inductive invariant and valid in the initial state – we can conclude that the safety property is an operational invariant for the specific system defined by its initial state and its specification of system behavior.

We have discussed in Example 1.1 (p. 4) that the system will likely violate the safety property: if the rules a2f and f2f do not have additional safeguards, nothing prevents a shuttle from switching to or remaining in speed mode fast just before arriving at a switch. With such safeguards in place, a shuttle would always need to brake when arriving at high velocity in the vicinity of a switch. Then, inductive invariant checking could prove preservation of the safety property by system behavior: a2f and f2f are not applicable when they would be leading to a violation and the other rules do not result in a fast shuttle anyway.

As explained on a general level above, the current approach to 1-inductive invariant checking does not allow verification of an arbitrarily large set of initial states – i.e. different track topologies with shuttle placement – in one pass. Similarly, its inductive step does not take invariant information derived from common properties of these initial states into account. Here, depending on the scenario, this information may include non-existing topology variations or an upper (or lower) bound on the number of shuttles. △

From the examples presented earlier and above (Example 1.10) and from considerations in earlier work [BLD+11, GL12, Dyc12], we can identify three key points that can be improved in comparison to the state of the approach and its formalization [Dyc12]:

State space restrictions. There may be implicit knowledge that should be considered by the verification approach. Reasons may be rooted in external assumptions that are not part of the system's specification, in properties established by previous verification procedures, or in physical properties that should be accurately reflected in the state space. Alternatively, we may want to deliberately restrict the state space of a system and have the restrictions reflected during symbolic verification.

Infinite number of inital states. In order to support systems with a possibly infinite set of initial states, we need to extend the verification approach to analyze infinitely many initial states in a finite fashion.

Completeness and false negatives. While the approach should remain terminating and sound, we would like to find ways to reduce the number of false negatives and hence, improve the degree of completeness.

Example 1.11. Given the three points above, here are some exemplary occurrences in the three main examples presented earlier:

Regarding the restriction of state spaces and additional information to be taken into account, Example 1.10 (p. 12) has already mentioned an upper or lower bound on the number of shuttles or restrictions on track topology, which are otherwise only reflected in the set of initial states. Depending on the formal model, absence of physical impossibilities – e.g. a shuttle existing on multiple tracks – may also have to be modeled in this fashion. For the model transformation examples, well-formedness of source (or target) models is a typical example: we may expect to have only well-formed input models (which could also be verified separately).

The cardinality of initial states has already been discussed and summarized in Table 1.1 (p. 7). We would like to allow an arbitrarily large set of track topologies for the shuttle protocol. Regarding behavior preservation of model transformations, all source and target models connected by the transformation should be considered (Example 1.3 (p. 7)).

The case of incompleteness and false negatives is difficult to discuss without delving into the formal and technical details of the approach. For the shuttle protocol, one reason for the occurrence of false negatives lies in the locality of the analysis. If we analyze only singular behavioral steps, we can only observe changes in a shuttle's speed mode between two values, not consider the evolution of its speed mode over more than one step. However, depending on the system's behavioral specification, speed mode evolution over a series of steps may be crucial when employing safeguards intended to prevent violations of the safety property. Thus, 1-inductive invariant checking may not be sufficient. △

1.3. Contribution: Extending 1-Induction and k-Induction

In order to implement the three points listed above, we first extend our basic and generic Verification Problem 1.1 (p. 1). For the first point, we need to take restrictions on state spaces into account:

Verification Problem 1.2. *Given a system defined by a system metamodel, an initial system state, specification of system behavior, and restrictions on the state space and given a set of safety properties, does every state in the restricted state space of the system satisfy the safety properties?*

Then, we can allow for the specification of a set of initial states instead of just one such state:

Verification Problem 1.3. *Given a set of systems defined by a system metamodel, a set of initial states, specification of system behavior, and restrictions on the state space and given a set of safety properties, does every state in the restricted state spaces of all systems satisfy the safety properties?*

That leaves the requirement of reducing the number of false negatives and improving completeness of the approach. As has become apparent in Example 1.11, one reason for the incompleteness is the local character of the analysis: only one step of system behavior is considered, leaving the larger evolution and context of a potential safety violation aside.

This thesis's approach of choice to extend the capabilities of inductive invariant checking with respect to context information is *k-induction*. Instead of 1-inductive invariants – properties whose validity in one state imply their validity in a subsequent state – we verify *k-inductive invariants*:

Definition 1.12 (k-inductive invariant)**.** *A* k-inductive invariant *is a property whose validity in a path of $k - 1$ system states implies its validity in any subsequent state after the execution of a single step of system behavior.*

As such, k-inductive invariants are a generalization of 1-inductive invariants and the latter are a special case of the former.: Given $k = 1$, a k-inductive invariant is a property whose validity in a path of length 0 – i.e. in a single state – implies its validity in subsequent states. This is equivalent to the definition of 1-inductive invariants.

The key idea behind k-induction, or k-inductive invariants, is to accumulate additional information by taking into account a path of rule applications instead of just one behavioral step. The approach to inductive invariant checking sketched in the previous section uses a symbolic encoding that represents all minimal contexts leading to a potential violation. Considering more than one step that leads to a violation extends this context and the chance that its additional information allows us to discard potential counterexamples. We may also find out that the violation in question is not reachable given our specification of system behavior.

Replacing 1-induction by k-induction does not only change the notion of inductive invariants used in the inductive step, it requires adjusting the base case to. Applying a k-inductive invariant as the inductive step of an inductive argument requires validity of the property in a path of $k - 1$ states in the first place as the base case. Hence, with respect to our notions of systems, state spaces, and behavioral steps, an inductive invariant using k-induction consists of the following parts:

1. the base case, where a property is shown for all paths of length $k - 1$ from a system's initial states and
2. the inductive step, where it is shown that the property is a k-inductive invariant.

In particular, if we want to take a set of initial states into account, the base case has to show validity of the property in all paths of the respective lengths from all possible initial states. Restrictions of the state spaces to be analyzed, on the other hand, might enable us to omit analyzing certain paths of behavioral steps that do not conform to the restrictions.

Similar to 1-induction, k-induction is not specific to graph transformation systems. It has been used in verification of finite state machines [SSS00] and in software verification [DHKR11]. The aim of this thesis is to formalize and implement verification with k-inductive invariants for graph transformation systems by extending the existing approach for 1-inductive invariants [BBG+06, Dyc12].

With respect to the shuttle protocol from Examples 1.1 (p. 4) and 1.10 (p. 12) and the application of 1-induction or k-induction, consider the following example:

Example 1.13. In Example 1.10 (p. 12), 1-inductive invariant checking only yields a positive result if shuttles can somehow detect that the subsequent track is a switch directly before moving to it. However, it can only decrease its speed mode by one step before reaching the switch. If we require an even lower speed mode on switches (i.e. slow), shuttles should be notified of an upcoming switch earlier – for example, two tracks ahead.

However, 1-inductive invariant checking only considers the last step before a violation occurs – and only considers its local context. We would find a counterexample where the shuttle moves from one track to the subsequent track – the switch. Since the analysis does not consider what happened before, the fact that the shuttle is notified of the upcoming switch is not considered: our system may actually be safe even if 1-inductive invariant checking claims otherwise. Taking into account a second behavioral step will reveal that.

In other words, verification with a 2-inductive invariant would have succeeded. The inductive argument behind this reasoning requires the following base case and inductive step:

Base case: in the initial states and all subsequent states – i.e., all paths of length 1 – there is no shuttle located on a switch in a speed mode other than slow.

Inductive step: no sequence of two behavioral steps results in a fast, braking, or accelerating shuttle positioned on a switch – unless that situation was already present before the first or the second rule application.

Then, by inductive argument, we can successfully verify the safety property as an operational invariant – i.e. for the entire state space – of systems consisting of the specified behavior and the set of initial states. If there is just one initial state, we need only to analyze all relevant paths from that initial state in the base case. Taking restrictions of the state space into account (our third requirement) might involve discarding certain behavioral steps or paths from initial states during the analysis. △

Even if we had just one initial state, the example above reveals another argument for the symbolic verification of the base case. If the initial state changes, it is no longer sufficient to verify the safety property in the new initial state. Instead, we have to check the initial state and all paths of the appropriate lengths originating from it. In other words, we have to perform explicit-state model checking with a bounded state space of one $k-1$ behavioral steps. While this will still be feasible in reasonable time for many systems, increasing the value of k for the induction will further increase the model checking effort. In the worst case, the number of bounded paths from the initial state rises exponentially with their length. Allowing our verification approach to show the base case of its inductive argument in a symbolic fashion addresses this problem – and allows for the specification of sets of initial states instead of just one initial state at a time.

Lastly, we need to consider the problem of computational effort. With a highly expressive formalism, we may encouter undecidable problems. Even with restrictions, termination may not be guaranteed – and if it is, computational effort may well be beyond a reasonable scope, depending on the size and complexity of the systems in question. Without delving into details, consider the combinatorial complexity of increasing the value of k from n to $n+1$: each rule needs to be combined with each path of length n already created – and there may be exponentially many possible combinations per pair of rule and path. As a result, we need to implement our approach to k-inductive invariant checking in reasonably efficient algorithms. This may require imposing restrictions on the formalism of graphs and graph transformation systems used to specify systems and system behavior, similar to the restricted formal model described in earlier work [Dyc12].

In summary, this thesis builds on a formalization, implementation, and application of 1-inductive invariant checking for graph transformation systems using a restricted formal model. It focuses on extending the approach on all three levels towards supporting restrictions on state spaces and infinite sets of initial states, both reflected in Verification Problem 1.3 (p. 13), and improvements in completeness by using k-induction instead of 1-induction. As such, its contribution consists of the following elements:

Formal-general – the description and justification of a formal and symbolic approach solving Verification Problem 1.3 (p. 13) by verification of graph transformation systems with k-induction, called the *general approach*,

Formal-restricted – the description and justification of a formal and symbolic approach solving Verification Problem 1.3 (p. 13) by verification of graph transformation systems with k-induction for the restricted formal model, called the *restricted approach*,

Impl.-restricted – the implementation of the approach described by **Formal-restricted** as an automated procedure,

where the formal approach (**Formal-restricted**) and implementation (**Impl.-restricted**) are applicable to meaningful scenarios, and, in their application, provide a positive result or meaningful symbolic counterexamples and have the following properties:

Appl.-soundness – soundness,
Appl.-termination – termination,
Appl.-deg.completeness – a reasonable degree of completeness,
Appl.-performance – reasonable performance.

In order to avoid confusion of 1-inductive and k-inductive invariants (and 1-induction and k-induction), in the following, all instances to the former will be explicitly labeled as 1-inductive invariants. References to the latter will be labeled as k-inductive invariants or simply inductive invariants. Note that a 1-inductive invariant is also a k-inductive invariant for the special case of $k = 1$.

This thesis refers to and relies on a number of publications that have appeared in the context of author's work on inductive invariant checking for graph transformation systems. They are listed here in chronological order:

- In 2015, the author of this thesis and Giese *[1, 2]* used partial negative application conditions in order to improve performance of the (1-inductive) approach by reducing combinatorial complexity for certain elements of the algorithm. The formalization also allowed restricting systems' state spaces.
- Also in 2015, the author of this thesis et al. *[7]* modified an earlier approach for verification of behavior preservation for relational model transformations via (1-inductive) invariant checking [GL12] to also support operational model transformations.
- In 2017, the author of this thesis and Giese *[3, 4]* described an approach, formalization, and implementation for k-inductive invariant checking for graph transformation systems based on the restricted formal model used in earlier work [Dyc12].
- In 2018, the author of this thesis, Giese, and Lambers *[6, 5]* extended earlier results for verification of behavior preservation of model transformations [GL12] to cover more complex cases and to improve the degree of automation. Verification was performed by 1-inductive invariant checking.

All publications by the author of this thesis that have appeared in the context of this thesis will be referred to in numerical style and in italics. All other publications, including older publications by the author, will be cited following alphanumerical style.

1.4. Outline

The remainder of this thesis is structured as follows:

In Chapter 2, we will reintroduce the formal framework of graphs and graph transformation systems. The formal model used in this thesis to describe systems and their specifications is based on core concepts of these formalizations. Chapter 2 will also revisit important construtions, which will later be used in the algorithmic implementation of this thesis's verification approach.

Given a formal view on system specification based on graphs and graph transformation systems, Chapter 3 will outline the basic principles and proof obligations of verification with 1-inductive invariants described in earlier work [BBG+06, Dyc12].

Chapter 4 will make the first step towards this thesis's contribution: it will provide the formal basis of extending the existing appoach with respect to restrictions on state spaces, sets of initial states, and k-inductive invariant checking. Similar to Chapter 3, it will formally reason about the basic principles and proof obligations of the inductive argument, i.e. the base case and the inductive step. While it will not discuss the algorithmic perspective or implementation of the approach, this chapter is the basis for all elements of this thesis's contribution (cf. Section 1.3).

Chapter 5 will take up the proof obligations established in Chapter 4 and describe the general approach to k-inductive invariant checking, thusly realizing **Formal-general**. It introduces a symbolic encoding capable of representing a possibly infinite number of transformation sequences in a finite fashion and describes a way to construct these representations. It also describes how representations are analyzed symbolically in order to establish the base case and inductive step of the inductive argument described in Chapter 4.

Chapter 6 will mimic Chapter 5 for the restricted approach to k-inductive invariant checking. After reiterating the restricted formal model used in earlier work, it will revisit and refine the definitions, constructions, and theorems of the general approach discussed in Chapter 5, thusly realizing **Formal-restricted**. Given the mathematical realization of the restricted approach, we will prove its soundness (**Appl.-soundness**) and argue that the constructions involved are finite (**Appl.-termination**). Furthermore, this chapter will describe the implementation – **Impl.-restricted** – of the restricted approach from an algorithmic perpective using pseudocode.

In Chapter 7, a number of extensions to the restricted approach will address some of its limitations. The extensions will mostly focus on the properties **Appl.-performance** and **Appl.-deg.completeness** of the contribution.

Chapter 8 will focus on related approaches for symbolic verification of graph transformation systems that are comparable to inductive invariant checking. Approaches will be discussed and compared on a theoretical basis.

Chapter 9 will discuss and evaluate the restricted approach and its implementation (Chapters 6 and 7), particularly with respect to **Appl.-performance**, **Appl.-deg.completeness**, and the applicability of the approach. It will also offer a more practical perspective on some of the results of Chapter 8.

Finally, Chapter 10 will summarize the findings of this thesis and its results and contribution with respect to its formal approach, implementation, and evaluation, and will provide an outlook to future work related to the verification of graph transformation systems with k-induction.

2. Prerequisites: Formal Foundations

Formal verification of complex systems requires a formalism systems can be mapped to. In the context of this thesis, this formalism will be typed graphs and typed graph transformation systems. In this chapter, we reiterate well-established formalisms from the literature [EEPT06, EGH$^+$14, HP09, Pen09]. In summary, we employ typed graph transformation systems with nested application conditions and (nested) graph constraints – with the special restriction that subconditions in application conditions and constraints may consist of injective morphisms only. With respect to graph rules, we follow the double-pushout approach with injective rule matching and use graph grammars to describe specific systems.

This leads to a *basic formal model*, where elements of systems and their specifications (Chapter 1) are mapped to their formal equivalents as follows:

Formal Model (basic)**.** *Systems and system specifications consist of the following elements:*

System metamodels *are specified by type graphs.*

System states *– including initial states – are described by typed graphs.*

System behavior *is described by a typed graph transformation system, which consists of typed graph transformation rules.*

Properties *are modeled as graph constraints.*

Systems *are specified by typed graph grammars, which consist of an initial state – a start graph – and a typed graph transformation system.*

System state spaces *are described by the state spaces – the set of all reachable graphs – of the corresponding graph grammars.*

All concepts and their prerequisites will be introduced in Section 2.1.

While this thesis focuses nearly exclusively on typed graphs, there is extensive research on the classification of graphs and graph morphisms in the general framework of category theory [EEPT06, EGH$^+$14, HP09]. The category of typed graphs with the class of typed injective graph morphisms belongs to the group of $\mathcal{M}$*-adhesive categories*, which conveys a number of properties useful in proofs and constructions [EGH$^+$14]. While some of this thesis's results may be generalizable beyond the category of typed graphs, such generalizations are beyond the scope of this thesis. Where possible, the formalisms used herein will stay clear of category theory and remain within the boundaries of typed graphs and typed graph morphisms.

Outline. This chapter is structured as follows: In Section 2.1, we will introduce the formal foundations of graphs and graph transformation systems, following well-established definitions [EEPT06, EGH$^+$14, HP09, Pen09]. Section 2.2 contains lemmas and constructions for nested application conditions and graph constraints. These constructions will be important elements of our verification algorithms. With respect to this thesis's contribution, Sections 2.1 and 2.2 introduce the formal foundations our general approach and elements of our restricted approach to k-inductive invariant checking will be based on.

2.1. General Foundations of Graph Transformation Systems

Graphs as the most fundamental concept of graph transformation systems consist of nodes and edges, with nodes representing specific entities and edges representing connections and associations between those entities. A common definition of a graph as a mathematical structure

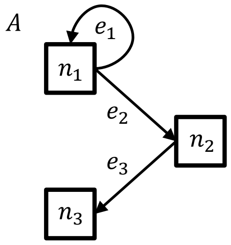


Figure 2.1. – An example graph A

uses sets that contain the graph's nodes and edges, and two functions describing how nodes are connected by edges.

Definition 2.1 (graph [EEPT06])**.** *A* graph $G = (V, E, s, t)$ *consists of a set of vertices* V*, a set of edges* E*, and the source and target functions* $s, t : E \to V$*, which map source and target nodes to edges.*

This definition allows multiple parallel edges between two nodes. It also requires edges to have a specific direction via the source and target functions. However, bidirectional edges can be represented by two edges with complementing source and target nodes. Furthermore, the definition allows loops – edges with identical source and target nodes.

Example 2.2 (graphs)**.** Consider the example graph A shown in Figure 2.1, where, following the formalism in Definition 2.1, we have $A = (V_A, E_A, s_A, t_A)$ with the set of vertices $V_A = \{n_1, n_2, n_3\}$, the set of edges $E_A = \{e_1, e_2, e_3\}$, and source and target functions $s_A, t_A : E \to V$. In particular, as can be inferred from the figure, $s_A(e_1) = n_1$, $s_A(e_2) = n_1$, and $s_A(e_3) = n_2$; furthermore, $t_A(e_1) = n_1$, $t_A(e_2) = n_2$, and $t_A(e_3) = n_3$. △

In order to describe how graphs and their elements relate to each other, we use the notion of graph morphisms.

Definition 2.3 (graph morphism [EEPT06])**.** *A* graph morphism $g = (g_V, g_E)$ *between two graphs* $G_1 = (V_1, E_1, s_1, t_1)$ *and* $G_2 = (V_2, E_2, s_2, t_2)$*, denoted as* $g : G_1 \to G_2$*, consists of two functions* $g_V : V_1 \to V_2$ *and* $g_E : E_1 \to E_2$*, which map nodes and edges, respectively, and preserve the source and target functions, i.e.* $s_2 \circ f_E = f_V \circ s_1$ *and* $t_2 \circ f_E = f_V \circ t_1$*.* G_1 *and* G_2 *are called the* domain *and* codomain *of* g*, respectively.*

Since functions (here: g_V, g_E) are required to map all elements from their domain, this definition requires graph morphisms to be *total* in the sense that all nodes and edges of the domain (graph) are mapped to elements in the codomain. For certain applications, it makes sense to drop this requirement and allow *partial morphisms* [Pen09], which use partial functions in order to map nodes and edges. Partial morphisms and their application will be introduced and explained in Section 7.3 of Chapter 7. While beneficial – in the context of this thesis – for reasons of performance and implementation, they are not required for the basic concept and theory of inductive invariant checking. Hence, unless noted otherwise, this thesis will assume graph morphisms to be total – i.e, based on the definition above.

Definition 2.4 (injective, surjective, and bijective graph morphisms [EEPT06])**.** *A graph morphism* $g = (g_V, g_E)$ *is* injective, surjective, *or* isomorphic*, if both* g_V *and* g_E *are injective, surjective, or bijective, respectively. Because of their foundation in category theory, injective (surjective, isomorphic) graph morphisms are also referred to as* monomorphisms *(*epimorphisms, isomorphisms*). An injective graph morphism* g *between two graphs* G_1, G_2 *is denoted as* $g : G_1 \hookrightarrow G_2$*.*

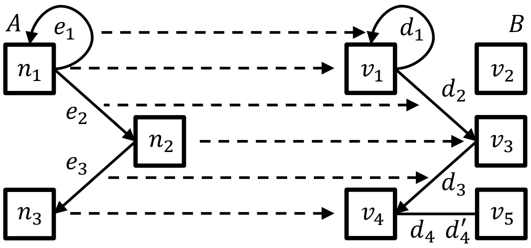


Figure 2.2. – Graph morphism $f : A \to B$

Example 2.5 (graph morphisms)**.** Figure 2.2 shows two example graphs $A = (V_A, E_A, s_A, t_A)$ and $B = (V_B, E_B, s_B, t_B)$ with $V_B = \{v_1, v_2, v_3, v_4, v_5\}$ and $E_B = \{d_1, d_2, d_3, d_4, d_4'\}$. Note that the edge between v_4 and v_5 is an undirected edge represented by two edges d_4, d_4' with $s_B(d_4) = v_4$, $t_B(d_4) = v_5$, $s_B(d_4') = v_5$, and $t_B(d_4') = v_4$.

A and B are connected by a graph morphism $f = (f_V, f_E) : A \to B$. In particular, f_V maps all nodes from A to nodes from B as follows: $f_V(n_1) = v_1$, $f_V(n_2) = v_3$, and $f_V(n_3) = v_4$. Likewise, f_E maps edges from A to edges from B: $f_E(e_1) = d_1$, $f_E(e_2) = d_2$, and $f_E(e_3) = d_3$. Since no two nodes or edges from A are mapped to the same element in B, f is injective and can therefore be denoted as $f : A \hookrightarrow B$. However, since there are nodes and edges in B that do not have a preimage in A, f is neither surjective nor bijective. △

Usually, specific mappings between nodes and edges by morphisms will be omitted in figures, as they can be unambiguously inferred by the elements' positioning or their identifiers. Similarly, we will usually not specifically designate injective morphisms (through special arrows) in concrete examples; unless otherwise noted, all morphisms in concrete examples will be injective.

For every graph, there is an (injective) morphism with the graph as its codomain and the empty graph as its domain:

Fact 2.6 (empty graph and morphisms [EEPT06])**.** *For each graph G, there is an injective morphism $i_G : \varnothing \hookrightarrow G$.*

We can compose graph morphisms to create a new graph morphisms:

Fact 2.7 (composition of graph morphisms [EEPT06])**.** *Given two graph morphisms $g_1 : G_1 \to G_2$ and $g_2 : G_2 \to G_3$ with $g_1 = (g_{V_1}, g_{E_1})$ and $g_2 = (g_{V_2}, g_{E_2})$, the composition $g_2 \circ g_1 = (g_{V_2} \circ g_{V_1}, g_{E_2} \circ g_{E_1})$ is a graph morphism $g_2 \circ g_1 : G_1 \to G_3$.*

Injective (surjective, isomorphic) graph morphisms are closed under composition.

Finally, we can use a distinguished graph and graph morphisms to introduce typed graphs.

Definition 2.8 (type graphs, typed graphs, and typed graph morphisms [EEPT06])**.** *A* type graph *is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. A* typed graph *$(G, type)$ is a graph G with a* typing morphism *$type : G \to TG$.*

A typed graph morphism *between typed graphs $(G_1, type_1)$ and $(G_2, type_2)$ is a graph morphism $g : G_1 \to G_2$ that preserves the typing morphisms, i.e. $type_2 \circ g = type_1$.*

Example 2.9 (type graphs, typing morphisms)**.** Consider the example type graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ shown in Figure 2.3. TG consists of the node types $V_{TG} = \{\mathsf{Shuttle}, \mathsf{Track}\}$, the edge types $E_{TG} = \{\mathsf{next}, \mathsf{isAt}, \mathsf{slow}, \mathsf{acc}, \mathsf{fast}, \mathsf{brake}\}$, and source and target functions as shown in Figure 2.3. It is a representation of the metamodel of the shuttle protocol from Example 1.1 (p. 4). However, note that the type graph does not contain the metamodel's cardinalities.

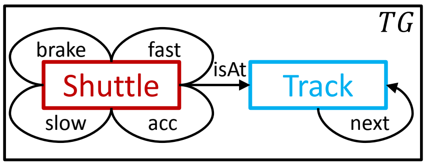


Figure 2.3. – Example type graph TG

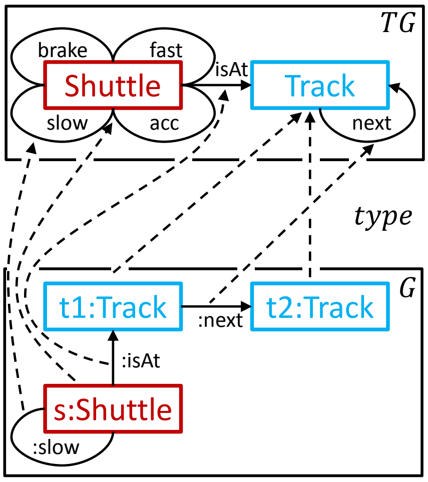


Figure 2.4. – Type graph TG, typed graph G, and typing morphism *type*

The intent behind this particular type graph is to formalize the shuttle protocol – our running example – using graphs and graph rules. Graphs typed over that type graph then represent states of our example systems. As before, tracks can be connected by directed edges of type next; the entirety of tracks and their connections can also be referred to as the track *topology*. Shuttles are positioned on tracks, which is denoted by edges of type isAt. Although actual shuttle movement is not decribed by a graph, system states have shuttles existing in different speed modes slow, acc, fast, and brake. Those speed modes are specified by loop edges of the respective type attached to the shuttles.

Figure 2.4 shows a graph G that is typed over TG by a (non-injective) typing morphism (depicted) $type : G \to TG$. In particular, G describes a shuttle in speed mode slow that is positioned on the first track of a topology of two (unidirectionally connected) tracks. △

Note that the types of nodes and edges in the example above can also be inferred by their names. Further example graphs will usually omit the type graph and typing morphism and denote node types as in this example, i.e. as ⟨identifier⟩ : ⟨type⟩. In some cases, especially for edges, the identifier will be omitted. Also, unless otherwise noted, all graphs, graph morphisms, and related notions appearing in this thesis will be typed.

Next, we introduce a core concept of graph transformation systems and an important contribution to their ability to represent complex systems: application conditions. Application conditions, as a generalization of negative [HHT96] and positive application conditions [EH86], are used to specify properties satisfiable by graph morphisms and, when appearing in a special form, by graphs.

Definition 2.10 (application conditions [EGH+14])**.** Application conditions*, or* nested application conditions *are inductively defined as follows:*

1. *For every morphism $a : P \to C$,* true *is an application condition over P.*

2. *For every morphism $a : P \to C$ and every application condition ac over C, $\exists(a, ac)$ is an application condition over P.*
3. *For application conditions ac, ac_i over P (with $i \in I$), $\neg ac$ and $\bigwedge_{i \in I} ac_i$ are application conditions over P.*

$$\begin{tikzcd} P \arrow[r, "g"] \arrow[d, "a"'] & G \\ C \lhd ac \arrow[ur, "q \models ac"'] \arrow[ur, phantom, bend left, "="] & \end{tikzcd}$$

Satisfiability of application conditions is inductively defined as follows:

1. *Every morphism satisfies* true.
2. *Given a morphism $a : P \to C$ and an application condition ac over C, a morhpism $g : P \to G$ satisfies $\exists(a, ac)$ if there exists an injective morphism $q : C \hookrightarrow G$ such that $q \models ac$.*
3. *A morphism satisfies $\neg ac$ if it does not satisfy ac. A morphism satisfies $\bigwedge_{i \in I} ac_i$ if it satisfies all ac_i.*

Two application conditions ac, ac' are equivalent, denoted $ac \equiv ac'$, if for all morphisms g, we have $g \models ac \Leftrightarrow g \models ac'$.
$\exists a$ appreviates $\exists(a, \text{true})$. $\forall(a, ac)$ abbreviates $\neg\exists(a, \neg ac)$.

More informally, this definition of satisfiability can be explained as follows: satisfiability of a simple existential condition (such as $\exists a$, see below on the left side) requires satisfying morphisms (g) and its image in the codomain (G) to be injectively extended (q) by the additional context supplied in the application condition's morphism's (a) codomain (C) while preserving the satisfying morphism's mapping, i.e. $g = q \circ a$. Conversely, a morphism satisfying a condition $\neg\exists a$ must not be extendable in this fashion.

$$\begin{tikzcd} P \arrow[r, "g"] \arrow[d, "a"'] & G \\ C \arrow[ur, "q"'] \arrow[ur, phantom, bend left, "="] & \end{tikzcd} \qquad \begin{tikzcd} P \arrow[r, "g"] \arrow[d, "a"'] & G \\ C \lhd ac \arrow[ur, "q \models ac"'] \arrow[ur, phantom, bend left, "="] & \end{tikzcd}$$

For more complex nested conditions (such as $\exists(a, ac)$, see above on the right side), the extending morphism (q) also needs to satisfy the nested condtion, i.e. $q \models ac$. For $\neg\exists(a, ac)$, a morphism satisfies the condition if all possible extensions do not satisfy ac.

Application conditions offer a means to establish context and complex logical conditions beyond a graph or graph morphism. A typical use case for application conditions is to further restrict the applicability of graph rules. Also, by means of satisfiability, application conditions implicitly represent the sets of morphisms satisfying them. When taking arbitrary graphs as codomains of such morphisms into account, those sets will, in general, be infinite. This cabability of symbolically encoding graph morphisms or graphs by application conditions will be an important part of the verification approach and algorithms at the core of this thesis.

Example 2.11 (application conditions)**.** Figure 2.5 shows two graphs P, C and a morphism $a : P \to C$ in an example application condition $\exists(a : P \to C)$ – or $\exists a$ for short. The node and edge mappings of a can be inferred by the positioning of the elements and by their identifiers. Informally, satisfying morphisms from P to another graph (say, G) are required to be extendable by mapping the additional elements in C to elements in G. Here, those

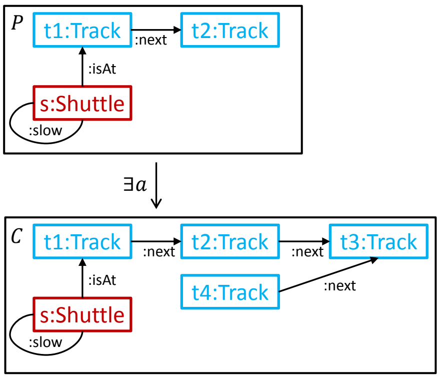
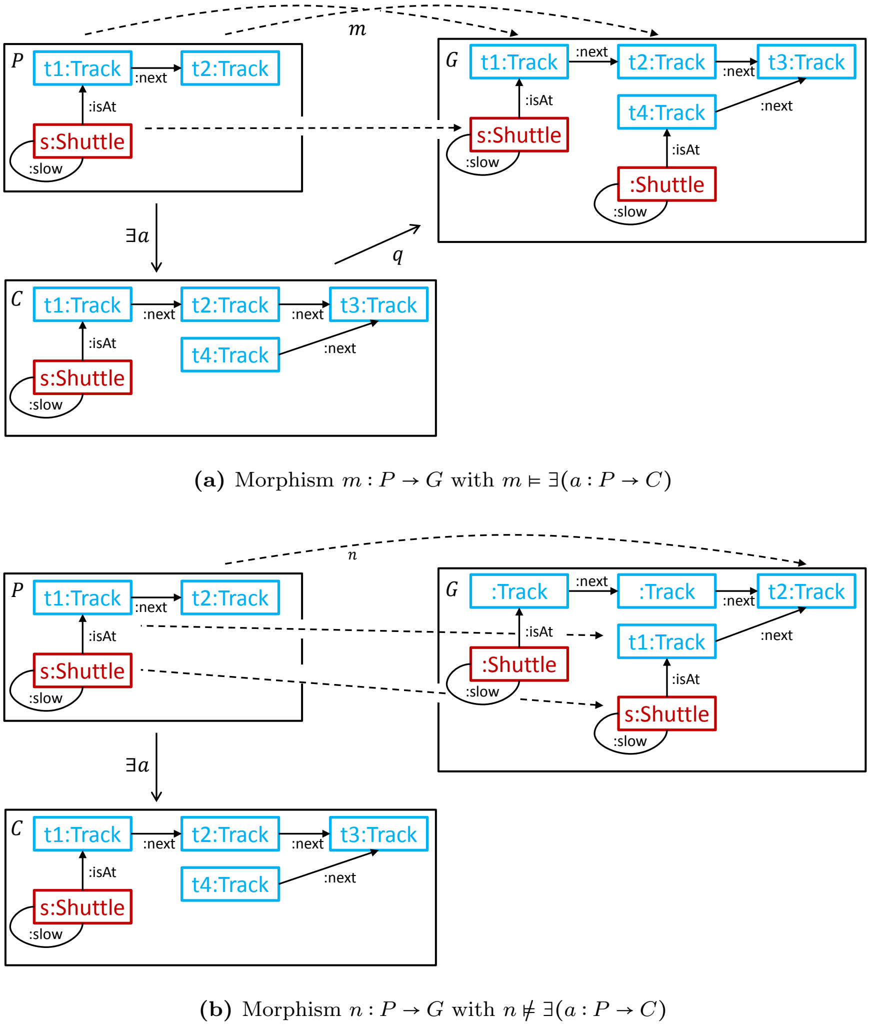


Figure 2.5. – Application condition $\exists(a : P \to C)$ with a graph morphism $a : P \to C$



(a) Morphism $m : P \to G$ with $m \models \exists(a : P \to C)$

(b) Morphism $n : P \to G$ with $n \not\models \exists(a : P \to C)$

Figure 2.6. – Example application condition and satisfiability

additional elements are two tracks (and their connections) which extend the tracks in P by a switch structure.

Figure 2.6(a) presents an example graph G and morphism $m : P \to G$ that satisfies $\exists a$. Consider the image of P under m in G: nodes t1, t2, s and their connecting edges. It is possible to find an injective morphism $q : C \hookrightarrow G$ that a) matches the image of P under a in C – t1, t2, s and connecting edges – to the aforementioned elements in G, i.e. $q \circ a = m$; and b) extends the image in G by the additional elements in C: the switch structure around the tracks t3 and t4.

Conversely, consider the morphism $n : P \to G$ shown in Figure 2.6(b). Here, n maps t1, t2, and s1 to a part of the switch structure. Because t2 does not have a subsequent track, there does not exist an injective (or any) morphism $x : C \hookrightarrow G$ that could map the additional tracks and their connecting edges to G while preserving n (i.e. while guaranteeing $x \circ a = n$). Hence, we have $n \nvDash \exists a$. By definition of satisfiability, that also implies $n \vDash \neg\exists a$ – and, for the first case, $m \nvDash \neg\exists a$.

We have seen in Example 1.1 (p. 4) that knowing and specifying not only a shuttle's position on a track and its subsequent track (as described in P), but also the existence of a switch beyond that makes sense from a safety perspective. Application conditions over us a way to do that.

Finally, consider the capability of application conditions to represent satisfying morphisms: the set of morphisms encoded by $\exists a$ contains m; similarly, $\neg\exists a$ represents n. While we will not find any more morphisms between P and G, there is an infinite number of other morphisms with P as their domain. Those morphisms, depending on whether or not they satisfy $\exists a$, would then also be represented by $\exists a$ or $\neg\exists a$. △

Whereas application conditions can be satisfied by graph morphisms, graph constraints describe properties satisfied by graphs. Formally, Graph constraints are a special type of application conditions:

Definition 2.12 (graph constraints [EGH+14])**.** *A* graph constraint*, or* nested graph constraint *is an application condition over the empty graph $\varnothing$. A graph G satisfies a graph constraint C, if the initial morhpism $i_G : \varnothing \hookrightarrow G$ satisfies C.*

Given a graph G, the definition of satisfiability requires the initial morphism i_G from the empty graph to G to satisfy the condition. Since every morphism, when composed with any initial morphism is isomorphic to any other initial morphism given matching domains and codomains, the question of satisfiability for any simple existential condition (such as $\exists i_C$, see below on the left) is reduced to the question of G containing C. In particular, any such inclusion will imply the existence of a morphism $q : C \hookrightarrow G$ that trivially satisfies $q \circ i_C = i_G$. Conversely, a condition $\neg\exists i_C$ is satisfied by a graph whenever it does not contain C as a subgraph.

$$\begin{tikzcd}
\varnothing \arrow[r, hook, "i_G"] \arrow[d, hook, "i_C"'] \arrow[dr, phantom, "="] & G \\
C \arrow[ur, hook, "q"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\varnothing \arrow[r, hook, "i_G"] \arrow[d, hook, "i_C"'] \arrow[dr, phantom, "="] & G \\
C \triangleleft ac \arrow[ur, hook, "q \vDash ac"'] &
\end{tikzcd}$$

Given a graph constraint with more complex nesting (such as $\exists(i_C, ac)$, above on the right side), the inclusion of C by q must also satisfy the nested condition, i.e. $q \vDash ac$. For $\neg\exists(i_C, ac)$, a graph G satisfies the constraint if all possible inclusions do not satisfy ac.

While application conditions can further restrict (and represent) graph morphisms, graph constraints apply this idea to graphs. Hence, a graph constraint is also a symbolic encoding for the set of graphs satisfying it. A common application of graph constraints is to specify modeling

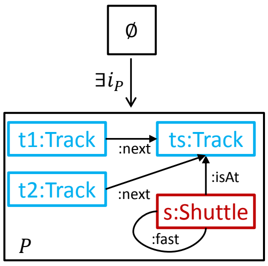
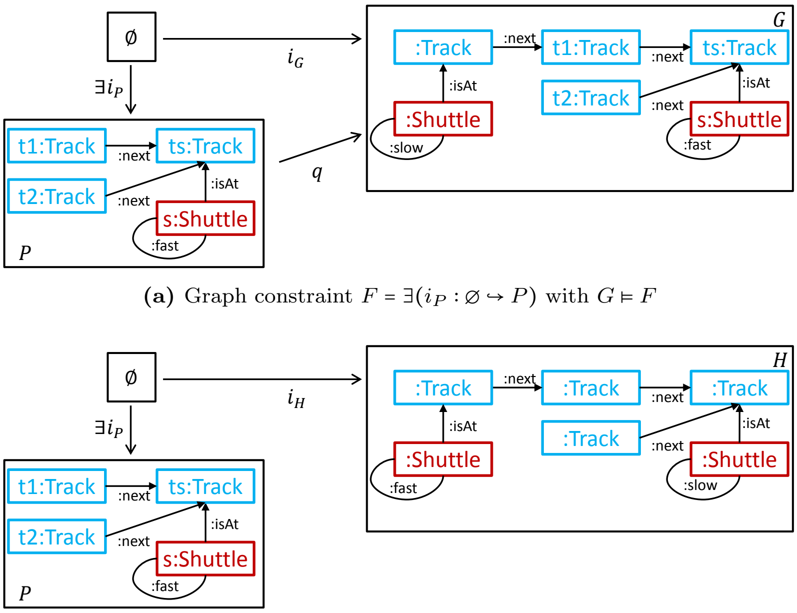


Figure 2.7. – Example graph constraint $F = \exists(i_P : \emptyset \hookrightarrow P)$



(a) Graph constraint $F = \exists(i_P : \emptyset \hookrightarrow P)$ with $G \vDash F$

(b) Graph constraint $F = \exists(i_P : \emptyset \hookrightarrow P)$ with $H \vDash \neg F$

Figure 2.8. – Example graph constraint and satisfiability

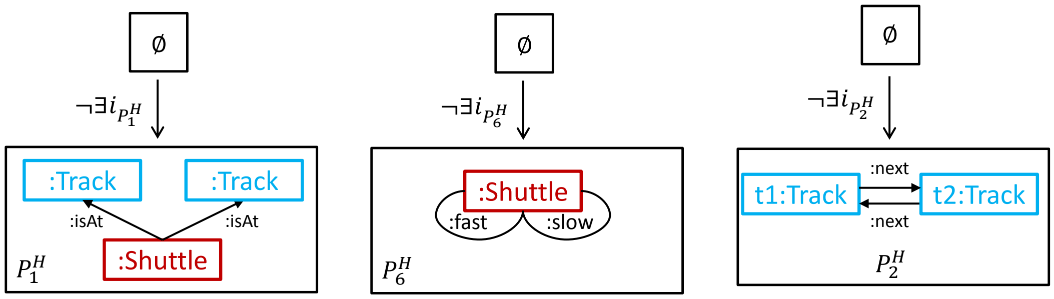
languages by combining a type graph with a graph constraint [GL12]. Then, the language is not only defined over typed graphs typed over the type graph, but is limited to those graphs also satisfying the additional graph constraint. In the context of this thesis, graph constraints are mostly used to a) model safety properties as forbidden subgraphs and b) specify additional constraints relevant to the system in question. The latter includes, for example, cardinality restrictions not modeled in type graphs.

Example 2.13 (graph constraints)**.** Figure 2.7 shows a graph constraint $F = \exists i_P$ (with i_P being the initial morphism $i_P : \emptyset \hookrightarrow P$). It specifies the existence of three tracks, two of which (t1 and t2) have the third (ts) as their subsequent track, which makes ts a switch. It also requires the existence of shuttle in speed mode fast and positioned on the switch track. As explained above, any graph containing P as a subgraph satisfies $\exists i_P$.

Figure 2.8(a) shows such a graph G that contains P as a subgraph. As depicted, there exists an injective morphism q including P into G by mapping t1, t2, ts and the shuttle to the respective elements in G. (Edges are mapped similarly.) Since $q \circ i_P = i_G$ holds trivially, i_G satisfies $\exists i_P$ and hence, by definition of satisfiability, G satisfies $\exists i_P$.

Note that there exists a second possible inclusion: t1 and t2 could be switched around while the mappings of ts and the shuttle remain the same.

Conversely, Figure 2.8(b) shows a different graph H where no such inclusion of P exists.



(a) Graph constraint $\neg H_1 = \neg\exists(i_{P_1^H} : \varnothing \hookrightarrow P_1^H)$

(b) Graph constraint $\neg H_6 = \neg\exists(i_{P_6^H} : \varnothing \hookrightarrow P_6^H)$

(c) Graph constraint $\neg H_2 = \neg\exists(i_{P_2^H} : \varnothing \hookrightarrow P_2^H)$

Figure 2.9. – Example graph constraints

While the tracks, shuttle, and next and isAt edges can be mapped as before, there is no matching edge for the shuttle's fast edge. The second shuttle lacks the positioning on a switch track required by P. Hence, H does not satisfy $\exists i_P$ (i.e. $H \nvDash \exists i_P$), which also implies $H \vDash \neg\exists i_P$ (or $H \vDash \neg F$), if we formulate the graph constraint as a negative.

A shuttle driving fast over a switch is a violation of the safety property introduced in Example 1.1 (p. 4); it might, for example, lead to the derailment of the shuttle. Then, $\neg F = \neg\exists i_P$ is the specifiaction of the property as a graph constraint, which should be satisfied by all graphs in the system's state space. Also, the constraint $\neg F$ represents all satisfying graphs, i.e. all safe states; conversely, F, i.e. $\exists i_P$, represents the (infinite) set of graphs where the safety property is violated.

Figure 2.9 also shows graph constraints; however, those constraints do not specify safety properties. Instead, Figure 2.9(a) models a cardinality constraint of the shuttle metamodel (Example 1.1, Figure 1.1 (p. 5)). The constraint in Figure 2.9(b) implements another restriction, although its equivalent in the metamodel would have to be specified as an OCL constraint. The same holds for $\neg H_2$ in Figure 2.9(c): here, tracks are required to be connected unidirectionally only – shuttles may not change directions. We will discuss the integration of these cardinality constraints and similar properties into system specifications and our verification approach in Chapter 4. △

With respect to application conditions (and graph constraints), there is one important restriction in our formalization: we require morphisms in subconditions to be injective. This has been described in the literature as *$\mathcal{M}$-normal form* [HP09, Pen09]. Since, in the category of (typed) graphs, $\mathcal{M}$ denotes the class of (typed) injective graph morphisms, we will introduce the notion as *injective normal form* below. For graph constraints, there is no difference in expressive power: every graph constraint not in $\mathcal{M}$-normal form can be equivalently transformed to $\mathcal{M}$-normal form [HP09]. For application conditions, this holds when satisfaction by injective morphisms only is concerned.

Definition 2.14 (injective normal form)**.** *An application condition ac is said to be in* injective normal form *if all morphisms in subconditions of ac are injective.*

In order to finally introduce graph rules and graph transformations, we first need to reiterate the concept of pushouts, which serve as a gluing construction. In particular, pushouts are a formalization for gluing two graphs together along a common subgraph [EEPT06].

Definition 2.15 (pushout [EEPT06])**.** *Given graphs and morphisms $f : A \to B$ and $g : A \to C$, a* pushout *(D, f', g') over f and g consists of a graph D and morphisms $f' : C \to D$ and $g' : B \to D$ such that $f' \circ g = g' \circ f$ and the following universal property is fulfilled: for all* *graphs H and morphisms $c : C \to H$, $b : B \to H$ with $c \circ g = c \circ f$, there is a unique morphism $q : D \to H$ such that $c = q \circ f'$ and $b = q \circ g'$.*

$$\begin{tikzcd}
A \arrow[r, "f"'] \arrow[d, "g"] \arrow[rd, phantom, "="] & B \arrow[d, "g'"'] \arrow[rdd, bend left, "b"] & \\
C \arrow[r, "f'"] \arrow[rrd, bend right, "c"'] & D \arrow[rd, "q"] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & \\
& & H
\end{tikzcd}$$

There is a construction to this declarative definition that allows creating pushouts for typed graphs by considering node and edge sets separately, then unambiguously determining source and target functions by the pushout property. In fact, pushout constructions for the category of (typed) graphs rely on the notion of pushouts in the category of sets [EEPT06].

Fact 2.16 (construction of pushouts [EEPT06])**.** *Given graphs $A_i = (V_i, E_i, s_i, t_i)$ for $i = 1..3$ and morphisms $f : A_1 \to A_2$ and $g : A_1 \to A_3$, we construct the pushout (D, f', g') over f and g as follows.*

$$\begin{tikzcd}
A_1 \arrow[r, "f"'] \arrow[d, "g"] \arrow[rd, phantom, "="] & A_2 \arrow[d, "g'"'] \\
A_3 \arrow[r, "f'"] & D
\end{tikzcd}$$

Construction. *We will construct node and edge sets of D separately:*

1. *We define the vertex set V'_D as $V'_D = V_2 \cup V_3$ and the relation $\sim_V$ as $\sim_V = \{(x, y) \mid x, y \in V'_D \wedge \exists v(f(v) = x \wedge g(v) = y)\}$.*
2. *We define $\equiv_V$ as the smallest equivalence relation containing $\sim_V$.*
3. *We define $V_D = \{[v] \mid v \in V'_D\}$, where $[v]$ is the equivalence class associated with v with respect to $\equiv_V$.*

1. *We define the edge set E'_D as $E'_D = E_2 \cup E_3$ and the relation $\sim_E$ as $\sim_E = \{(x, y) \mid x, y \in E'_D \wedge \exists e(f(e) = x \wedge g(e) = y)\}$.*
2. *We define $\equiv_E$ as the smallest equivalence relation containing $\sim_E$.*
3. *We define $E_D = \{[e] \mid e \in E'_D\}$, where $[e]$ is the equivalence class associated with e with respect to $\equiv_E$.*

We then define $f' = (f'_V, f'_E)$ and $g' = (g'_V, g'_E)$ as follows:

1. *$f'_V(v) = [v]$ for all $v \in V_3$.*
2. *$f'_E(e) = [e]$ for all $e \in E_3$.*
3. *$g'_V(v) = [v]$ for all $v \in V_2$.*
4. *$g'_E(v) = [e]$ for all $e \in E_2$.*

Finally, we need to define the source and target function s_D and t_D:

1. $s_D(e) = \begin{cases} g'(v_2) & \textit{if there exists } e' \in E_2 \textit{ such that } s_2(e') = v_2 \textit{ and } g'(e') = e \\ f'(v_3) & \textit{if there exists } e' \in E_3 \textit{ such that } s_3(e') = v_3 \textit{ and } f'(e') = e \end{cases}$
2. $t_D(e) = \begin{cases} g'(v_2) & \textit{if there exists } e' \in E_2 \textit{ such that } t_2(e') = v_2 \textit{ and } g'(e') = e \\ f'(v_3) & \textit{if there exists } e' \in E_3 \textit{ such that } t_3(e') = v_3 \textit{ and } f'(e') = e \end{cases}$

Then, (D, f', g') with $D = (V_D, E_D, s_D, t_D)$ is the pushout over f and g.

For the application of graph rules, we sometimes need the inverse construction to a pushout; i.e. given morphisms and a potential pushout graph, we need to find the pushout. This is described as a *pushout complement*:

Definition 2.17 (pushout complement [EEPT06])**.** *Given morphisms $f : A \to B$ and $g' : B \to D$, the* pushout complement *of f and g' is $A \to C \to D$ with $g : A \to C$ and $f' : C \to D$ if (D, f', g') – denoted as (1) – is a pushout over f and g.*

$$\begin{tikzcd}
A \arrow[r, "f"'] \arrow[d, "g"] \arrow[dr, phantom, "(1)"] & B \arrow[d, "g'"'] \\
C \arrow[r, "f'"] & D
\end{tikzcd}$$

For typed graphs, pushouts have a number of useful properties that will later be required for certain proofs and constructions.

Fact 2.18 (properties of pushouts [EEPT06])**.** *Pushouts have the following properties:*

$$\begin{tikzcd}
A \arrow[r, "f"'] \arrow[d, "g"] \arrow[dr, phantom, "(1)"] & B \arrow[d, "g'"'] \\
C \arrow[r, "f'"] & D
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & E \arrow[d] \\
C \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

1. *If f (above on the left) is injective (surjective), f' is also injective (surjective).*
2. *The pushout object D is unique (up to isomorphism).*
3. *If (1) and (2) (above on the right) are pushouts, (1) + (2) is a pushout.*
4. *If (1) and (1) + (2) are pushouts, (2) is a pushout.*

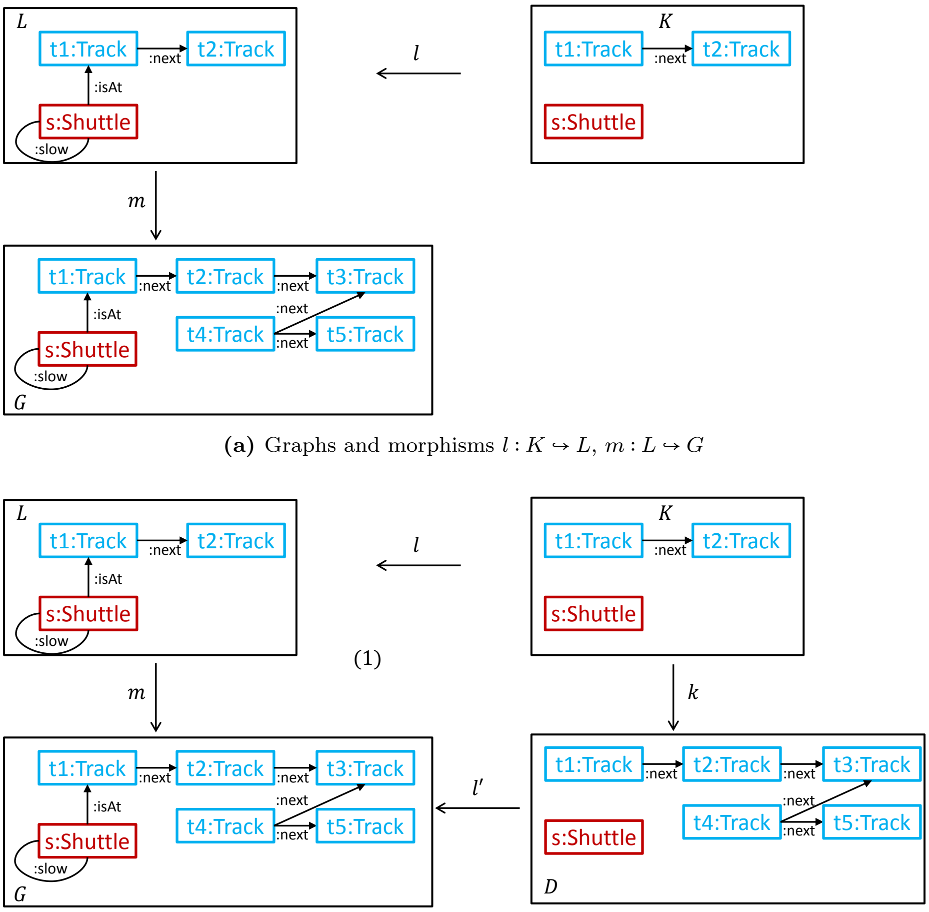
Example 2.19 (pushout complement, pushout)**.** Consider Figure 2.10(a) where, given morphisms $l : K \hookrightarrow L$ and $m : L \hookrightarrow G$, we are looking for the pushout complement of l and m. (1) in Figure 2.10(b) is a pushout (G, l', m) over l and k and hence, $K \hookrightarrow D \hookrightarrow G$ is the pushout complement of l and m.

Intuitively, D needs to supply the nodes and edges contained in G, but not in L: tracks t3, t4, t5, and the three next edges. D also needs to contain elements in G that have preimages in K under $m \circ l$: tracks t1 and t2, the shuttle s and one next edge. However, it must not contain the edges isAt and slow because those edges do not have preimages in K.

Given $k : K \hookrightarrow D$ as above and $r : K \hookrightarrow R$ (Figure 2.11(a)), we can construct the pushout over r and k, which is then depicted in Figure 2.11(b) as (H, r', m').

Here, as explained before, the intuition is that H is the result of gluing D and R together along the common subgraph K, where the subgraph relationships are specified by the morphisms r and k. Then, H needs to contain K – tracks t1 and t2, shuttle s and a next edge – and is extended by all elements in R and D that do not have preimages under r or k, respectively: tracks t3, t4, t5, next edges from D, and edges acc and isAt from R. The edges are attached to their source and target nodes via the graph's source and target functions according to their sources and targets in their original graphs and the nodes' images under m' or r', respectively. △

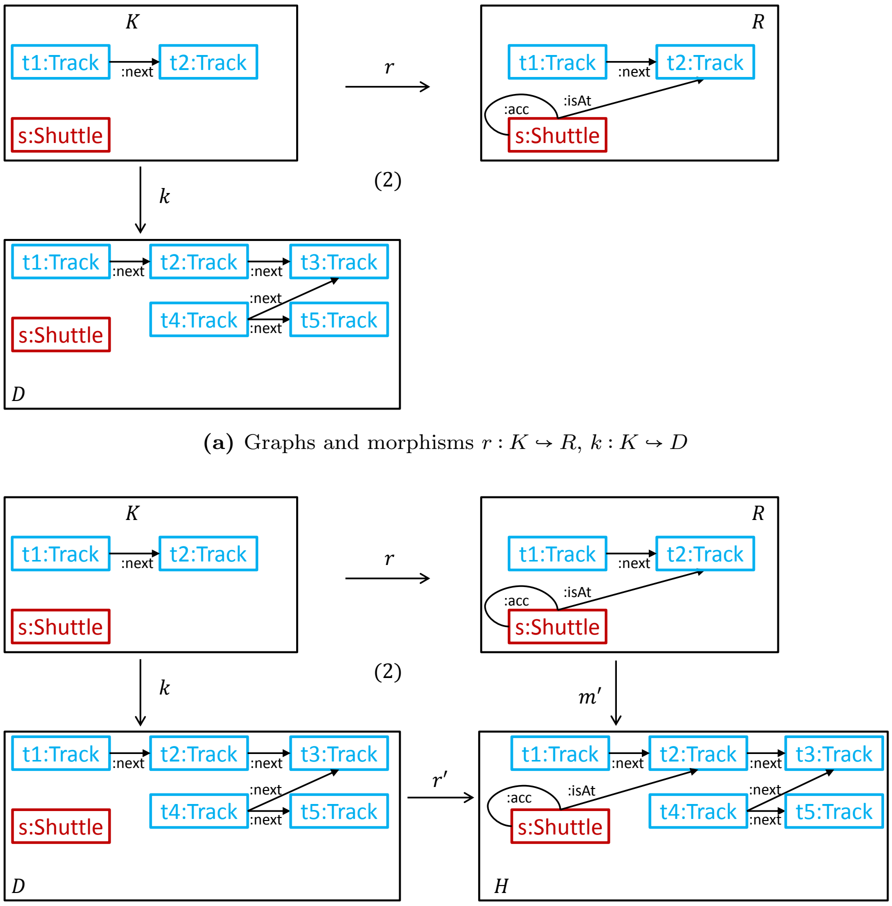
With all the prerequisites established, we are now ready to introduce graph transformation rules, or simply graph rules, as means of specifying behavior. Graph rules describe how graphs can be changed; the application of a graph rule (also called a graph transformation) to a graph



(a) Graphs and morphisms $l : K \hookrightarrow L$, $m : L \hookrightarrow G$

(b) Pushout complement $K \hookrightarrow D \hookrightarrow G$ of l and m

Figure 2.10. – Example pushout complement



(a) Graphs and morphisms $r : K \hookrightarrow R$, $k : K \hookrightarrow D$

(b) Pushout (H, r', m') over r and k

Figure 2.11. – Example pushout

then results in a new graph. More complex graph rules may contain application conditions, which allow more fine-grained control over the applicability of graph rules.

There are two well-established approaches for algebaric graph transformation: single-pushout (SPO) and double-pushout (DPO) [EEPT06]. This thesis focuses on the latter. The name double-pushout stems from the two pushout constructions required for the application of a graph rule as explained in the following definition.

Definition 2.20 (graph rule [EGH+14])**.** *A* plain graph rule $p = (L \hookleftarrow K \hookrightarrow R)$ *consists of two injective morphisms* $l : K \hookrightarrow L$ *and* $r : K \hookrightarrow R$*.* L *and* R *are called the left and right hand side of* p*, respectively. A* graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$ *consists of a plain rule and a* left *and* right application condition ac_L *and* ac_R*.*

$$\begin{tikzcd}
ac_L \rhd L \arrow[d, hook, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, hook', "l"] \arrow[r, hook, "r"'] \arrow[d] \arrow[dr, phantom, "(2)"] & R \lhd ac_R \arrow[d, hook, "m'"] \\
G & D \arrow[l, hook', "l'"'] \arrow[r, hook, "r'"] & H
\end{tikzcd}$$

The application of a graph rule, also called a graph transformation*, consists of two pushouts* (1) *and* (2) *(shown above) such that* $m \vDash ac_L$ *and* $m' \vDash ac_R$*. The transformation is denoted as* $G \Rightarrow_{b,m,m'} H$*. the morphisms* $m : L \hookrightarrow$ *and* $m' : R \hookrightarrow H$ *are called the match and comatch of* b *in* G *and* H*, respectively. We often write* $G \Rightarrow_b H$ *or* $G \Rightarrow H$ *to express that there exist* m*,* m' *(or* b*,* m*, and* m'*) such that* $G \Rightarrow_{b,m',m'} H$*.*

Given a set of rules $\mathcal{R}$*,* $G \Rightarrow_{\mathcal{R}} H$ *expresses that there exists a rule* $b \in \mathcal{R}$ *such that* $G \Rightarrow_b H$*.*

Note that this definition requires injective matches and comatches. This is not necessarily the case for common definitions of the concept; in particular, non-injective matches and comatches are often allowed, too [EEPT06, EGH+14, HP09]. For typed graph transformation systems, however, there is no difference in expressive power between both approaches [HP09]. This thesis focuses on injective rule matching.

While graph rules are commonly allowed to have both a left and right application condition, both concepts are equivalent in terms of expressive power [HP09]. Specifically, any right application condition can be transformed into an equivalent left application condition [HP09]. It can be argued that the specification of a left application condition, which is checked before rule application, is more intuitive than equipping rules with right application conditions, which have to be checked after rule application, possible invalidating the transformation. Hence, we will usually use application conditions in rules as left application conditions and will only use the trivial (and default) right application condition true.

Similar to the symbolic nature of application conditions and graph constraints, a rule can also be seen as a symbolic encoding for a number of graph transformations. Then, a rule b serves as a representation for a possibly infinite number of graph transformations $G \Rightarrow_b H$.

Fact 2.21 (application of graph rules [EGH+14])**.** *The successful application of a graph rule* $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ *to a graph* G *consists of the following conditions and steps:*

Find an injective match $m : L \hookrightarrow G$ *such that*

1. m *satisfies* ac_L
2. m *satisfies the dangling condition: no edge in* $m(L)$ *is adjacent to a node in* $m(L \smallsetminus l(K))$*.*

Then,

1. *Remove from* G *all items in* $m(L \smallsetminus l(K))$*, resulting in a graph* D*.*
2. *Add to* D *all items in* $R \smallsetminus r(K)$*, resulting in a graph* H *and the comatch* $m' : R \hookrightarrow H$*.*

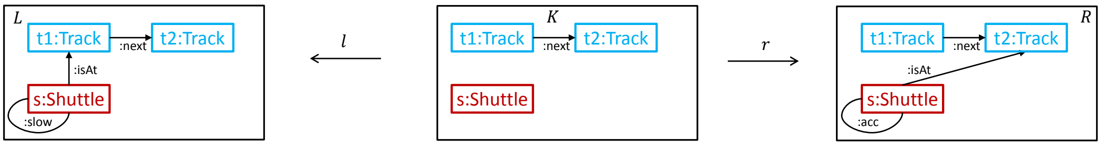


Figure 2.12. – Example graph rule $\mathsf{s2a}' = \langle (L \hookleftarrow K \hookrightarrow R), \mathrm{true}, \mathrm{true}\rangle$

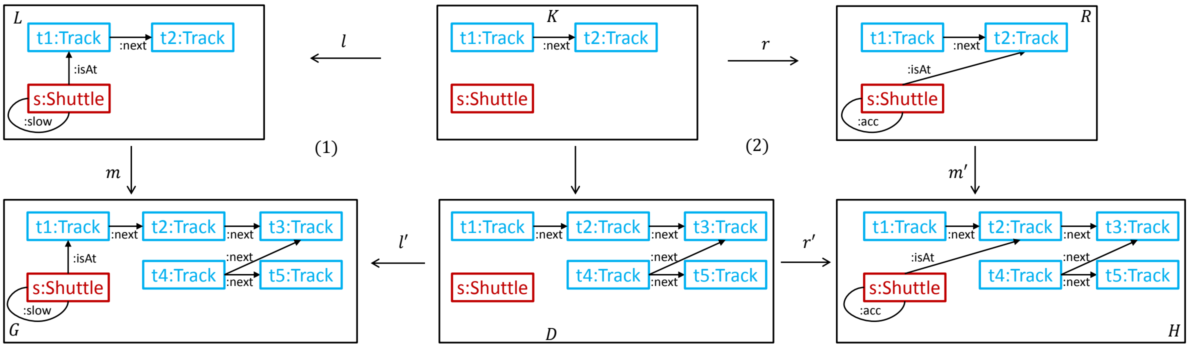


Figure 2.13. – Example transformation $G \Rightarrow_{\mathsf{s2a}',m,m'} H$

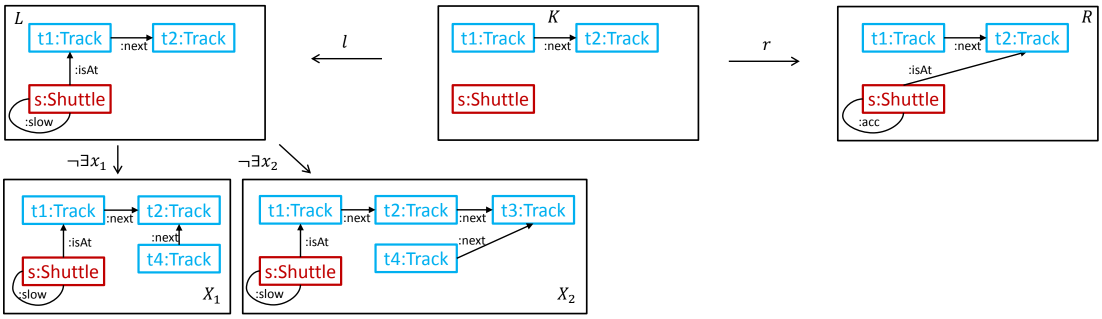


Figure 2.14. – Example graph rule $\mathsf{s2a} = \langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1 \wedge \neg\exists x_2, \mathrm{true}\rangle$

Note that the identification condition [EGH+14] is not needed here because we require injective rule matching.

Example 2.22 (graph rule)**.** Figure 2.12 shows a graph rule $\mathsf{s2a}' = \langle(L \hookleftarrow K \hookrightarrow R), \mathrm{true}, \mathrm{true}\rangle$ that models one aspect of our shuttle system's behavior. As expected, the rule describes a shuttle driving in speed mode slow moving to a subsequent track and changing its speed mode to acc(elerating). The injective morphisms l and r can be inferred by the elements' positioning and by their identifiers. The rule's left and right application conditions are simply true. Since any morphism satisfies true (Definition 2.10 (p. 22)), there are no further conditions on matches and comatches for the rule to be applicable.

Figure 2.13 shows a rule application $G \Rightarrow_{\mathsf{s2a}',m,m'} H$ of rule $\mathsf{s2a}'$ via a match $m : L \hookrightarrow G$ to a graph G. With the pushouts (1) and (2), this results in a graph H and comatch $m' : R \hookrightarrow H$. In particular, the shuttle s moves from track t1 to track t2 and changes its speed mode from slow to acc. Note that (1) and (2) correspond to the pushouts in Example 2.19 (p. 29).

Furthermore, Figure 2.14 shows the rule $(L \hookleftarrow K \hookrightarrow R)$ extended by an application condition $\neg\exists x_1 \wedge \neg\exists x_2$, which prevents the rule from being applied if there is a switch one or two tracks ahead of the moving shuttle. Since m (Figure 2.13) satisfies $\exists x_2$ and thus does not satisfy $\neg\exists x_2$, the rule $\langle(L \hookleftarrow K \hookrightarrow R), \neg\exists x_1 \wedge \neg\exists x_2, \mathrm{true}\rangle$ is not applicable to G via m. Moreover, since there does not exist another possible match of L in G, it is not applicable to G at all. This is an example of implementing safeguards in system behavior that may prevent the violation of safety properties; this idea has been mentioned in Example 1.10 (p. 12). △

Graph transformations are invertible by applying the inverse rule, which can be constructed as follows:

Fact 2.23 (inverse graph rule [EGH+14])**.** *Given a graph rule $b = \langle(L \hookleftarrow K \hookrightarrow R), ac_L, ac_R\rangle$, its* inverse graph rule *is $b^{-1} = \langle(R \hookleftarrow K \hookrightarrow L), ac_R, ac_L\rangle$. For every transformation $G \Rightarrow_{b,m,m'} H$, we also have a transformation $H \Rightarrow_{b^{-1},m',m} G$.*

Since the inversion of an inverse rule results in the original rule, the existence of a transformation $H \Rightarrow_{b^{-1},m',m} G$ also implies the existence of the transformation $G \Rightarrow_{b,m',m'} H$.

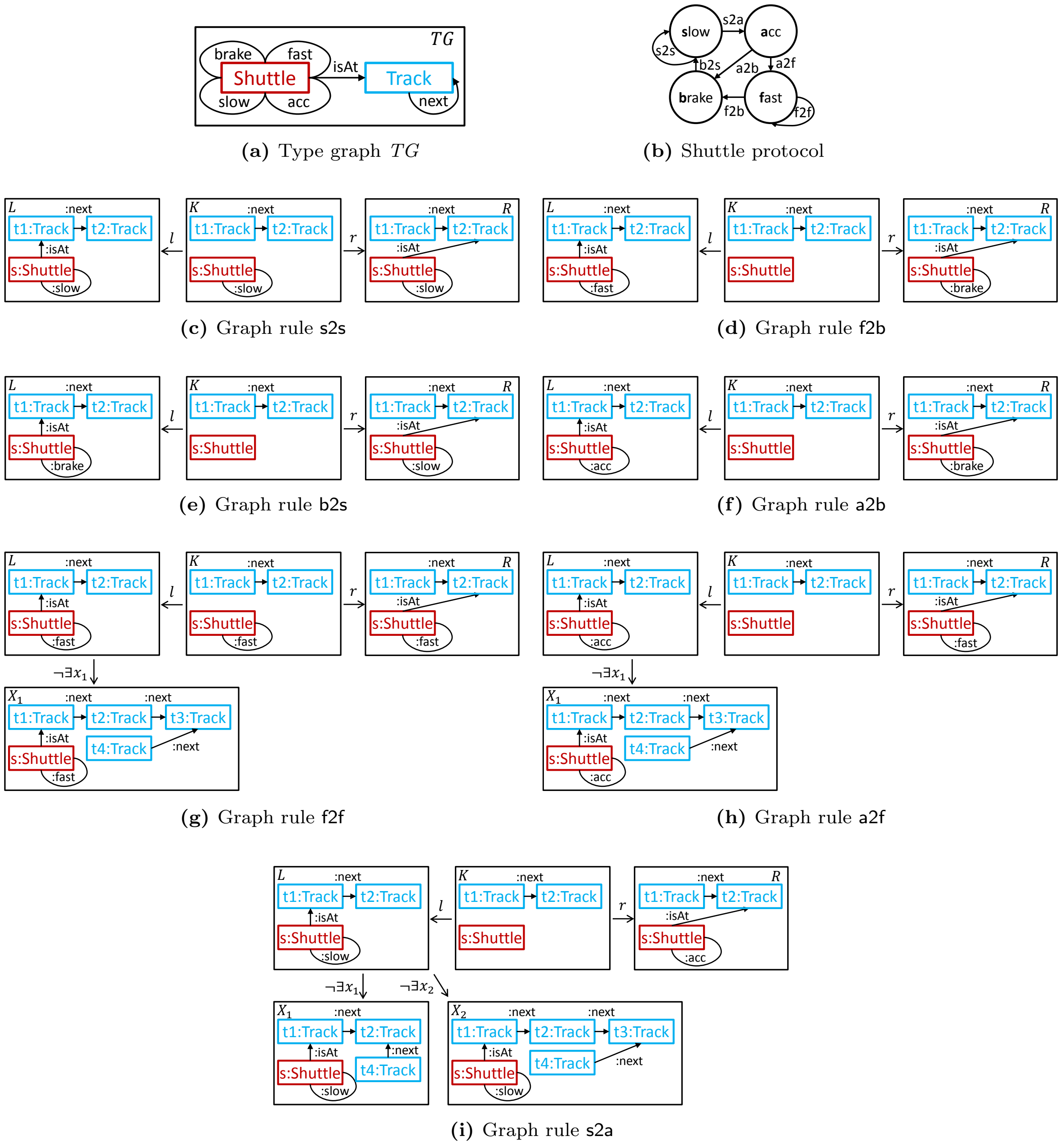
Multiple graph rules can be combined as a set of graph rules in a graph transformation system. A graph transformation system then specifies system behavior. Specific manifestations of said behavior occur in the form of rule applications to system states, i.e. to graphs. Since our focus in this thesis lies on typed graphs, we also have typed graph transformation systems:

Definition 2.24 (typed graph transformation system [EEPT06])**.** *A* typed graph transformation system *$GTS = (TG, \mathcal{R})$ consists of a type graph TG and a set of typed graph rules $\mathcal{R}$.*

Note that such a transformation system only supplies a specification of system behavior without specifying system states or, in particular, an initial state. Consequently, we cannot yet derive actual system execution paths or a state space for a particular system from such a specification.

Example 2.25 (graph transformation system)**.** Figure 2.15 shows an example graph transformation system $GTS = (TG, \mathcal{R})$ modeling the behavior of our running example – the shuttle protocol (Example 1.1 (p. 4)). As before, shuttles move along connected tracks in different speed modes. The system's type graph is shown in Figure 2.15(a). Figure 2.15(b) reiterates the protocol with respect to shuttles' speed mode changes; however, this is just a visualization and not an explicit element of the system.

All graph rules follow one of two basic structures: s2s (short for slow2slow) and f2f (short for fast2fast) do not change a shuttle's speed mode; s2a, a2b, a2f, f2b, and b2s do. Rules that



(a) Type graph TG

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

(f) Graph rule a2b

(g) Graph rule f2f

(h) Graph rule a2f

(i) Graph rule s2a

Figure 2.15. – Example graph transformation system [3, 4]

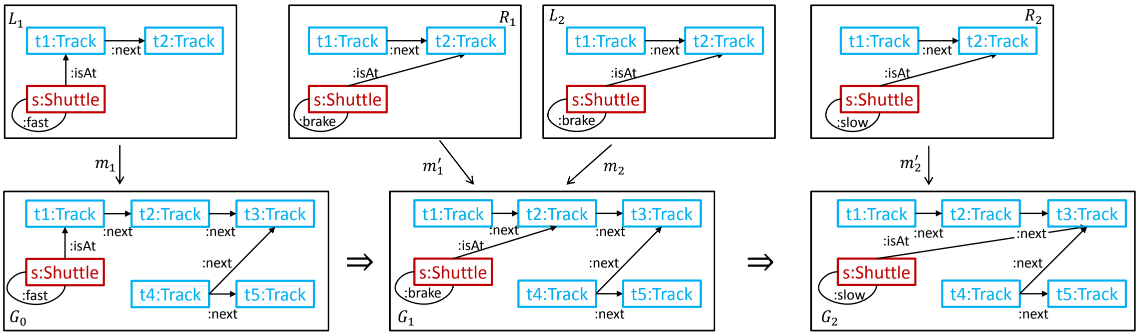


Figure 2.16. – Example transformation sequence *trans*

increase the shuttle's velocity or have the shuttle remain at high velocity (s2a, a2f, f2f) have additional safeguards in the form of application conditions. All rules include a shuttle moving from its current location to a subsequent track – speed mode transitions only occur in concert with shuttle movement. Note that even though the graph rules may have different left and right sides, we will reuse the identifiers L and R (and others) in examples. △

Next, we will introduce transformation sequences as a means to describe the subsequent application of graph rules to a graph via matches and comatches. Transformation sequences have intermediate result graphs and a final result graph. They can be used to describe traces in a system. For instance, it might be desirable to describe possible transformation sequences from a system's initial state or transformation sequences to violations of a safety property.

Definition 2.26 (transformation sequence [3]). *Given a set of graph transformation rules $\mathcal{R}$ and $k \geq 1$, a transformation sequence to $\mathcal{R}$ trans = $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_k$ denotes subsequent graph transformations $G_0 \Rightarrow_{\mathcal{R}} G_1$, $G_1 \Rightarrow_{\mathcal{R}} G_2$, ..., $G_{k-1} \Rightarrow_{\mathcal{R}} G_k$ with rules from $\mathcal{R}$. We say that trans has length k and also denote it by trans = $G_0 \Rightarrow_{\mathcal{R}}^{k} G_k$.*

When specific rules, matches, or comatches are of interest, we also write trans = $G_0 \Rightarrow_{b_1,m_1,m_1'} G_1 \Rightarrow_{b_2,m_2,m_2'} \cdots \Rightarrow_{b_k,m_k,m_k'} G_k$, with $b_i \in \mathcal{R}$ and m_i and m_i' being matches and comatches of the respective rules.

Example 2.27 (transformation sequence). Consider the transformation sequence *trans* = $G_0 \Rightarrow_{\mathsf{f2b},m_1,m_1'} G_1 \Rightarrow_{\mathsf{b2s},m_2,m_2'} G_2$ depicted in Figure 2.16. Note that the exact constructions (pushouts) of the two transformations have been omitted, but can be inferred. △

In contrast to the depiction in Figure 2.16, we will usually show transformation sequences in a more compact manner by leaving out left and right rule sides, matches, and comatches, unless they are of specific importance. Usually, those elements can be inferred from the graphs in the sequence and the names of the rules.

Note that a transformation sequence only describes one specific path of transformations and the situations involved. In any state space of a given system, there will usually be a large or infinite number of possible transformation sequences.

As explained before, graph transformation rules define system behavior in general terms, but do not describe actual transformations. For example, a graph transformation system does not have an initial state of a state space. Those concepts are included in the notion of graph grammars; in particular, a graph grammar can be seen as an instantiation of a graph transformation system by supplying an initial state. The entirety of possible system behavior in a graph grammar is then described by its state space.

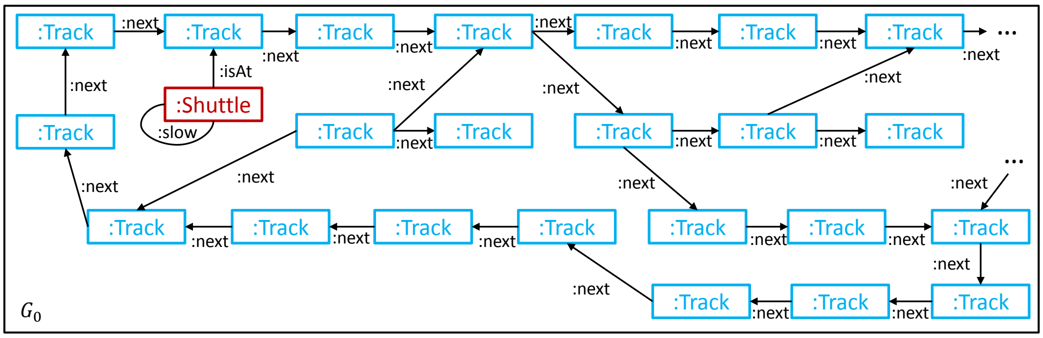


Figure 2.17. – Example initial graph G_0

Definition 2.28 (typed graph grammar [EEPT06], state space *[4]*)**.** *A typed* graph grammar *$GG = (G_0, GTS)$ consists of an* initial graph *or* start graph *G_0 and a typed graph transformation system GTS.*

The state space *of a graph grammar $GG = ((\mathcal{R}, TG), G_0)$ is defined as* REACH(GG) = $\{G \mid \exists n(G_0 \Rightarrow^n_{\mathcal{R}} G)\}$*, i.e. as the set of graphs reachable by graph transformations from the initial graph.*

Example 2.29 (graph grammar)**.** Figure 2.17 shows a fragment of a possible (finite) start graph of a graph grammar $GG = (G_0, GTS)$ with GTS as in Example 2.25 (p. 34). Beyond the part depicted, additional tracks – represented by the dots – exist, but the graph contains only one shuttle. The only graph rule applicable to the shuttle and the surrounding tracks is s2s. Since there are cycles (not depicted) in the track topology of G_0, there could be arbirtrarily long transformation sequences. However, because the track topology cannot be changed, the state space will be finite (up to isomorphism). In general, this is not necessarily the case. With rules adding tracks or shuttles to the system, infinite state spaces are very likely. △

Notation. For some of the concepts introduced above, Table 2.1 provides a quick reference over some notational guidelines. Most elements appearing in theorems, lemmas, constructions, and their proofs will follow the naming scheme shown in the table.

This concludes the discussion of formal foundations of graph transformation systems. Next, we will consider constructions and transformations applicable to graph constraints and application conditions.

2.2. Application Conditions and Constructions

As we have seen, application conditions and graph constraints are central elements of system specification in our approach. In particular, we employ graph constraints to specify safety properties and cardinality constraints; application conditions are used to restrict the applicability of graph rules.

However, application conditions can also be used to describe the applicability of a graph rule for a match of its left side with respect to the requirements for graph rule application (Definition 2.20 (p. 32) and Fact 2.21 (p. 32)). A potential match has to satisfy the rule's left application condition; in addition, the dangling condition has to be fulfilled. For the general case of weak adhesive HLR categories and non-injective matching, there is a theorem and construction to describe the creation of the required condition; the construction is called Appl [HP09]. Here, we use an adaption for typed graphs and graph rules with injective matching:

Table 2.1. – Overview of notation guidelines

Elements	Notation
Graphs	$A, B, D, E, G, H, L, P, Q, R, S, T, X$
Graph morphisms	$a, b, c, d, e, f, g, l, p, q, r, s, t, x, y$
Graph rules	b, c, ρ
Matches (comatches)	m (m')
Application conditions	ac
graph constraints	$C, F, H, SC, \mathcal{F}, \mathcal{H}, \mathcal{S}$
indices	$i, j, k, n, o, u, v, w, z$
index sets	$I, J, K, N, O, U, V, W, Z$

Lemma 2.30 (rule applicability condition)**.** *For every rule* $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \text{true}\rangle$*, there is a construction* Appl *such that for every injective morphism* $m : L \hookrightarrow G$*,* $m \vDash \text{Appl}(b) \wedge ac_L$ *if and only if there is a transformation* $G \Rightarrow_{b,m} G'$*.*

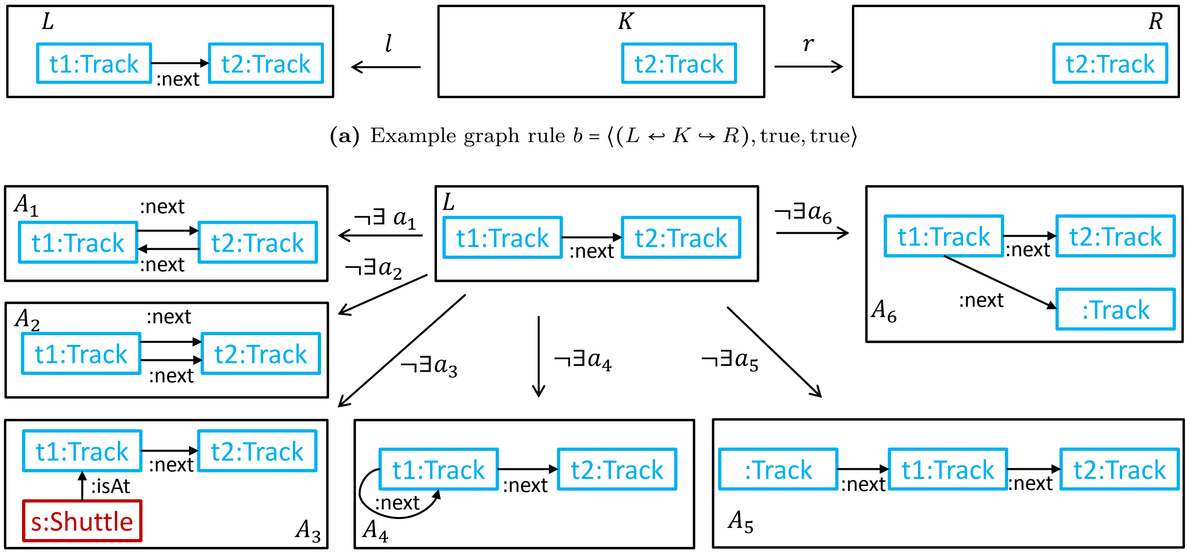
For specifics of the construction, we refer to the original source [HP09]. Intuitively, the procedure is as follows:

For each node v in L without a preimage in K under l – i.e. for each v in the set of nodes to be deleted by application of b – and for each edge type adjacent to the node type of v, i.e. for each edge type e_{TG} of edges possible adjacent to v, create the application conditions $\neg\exists d$ with injective morphisms $d : L \hookrightarrow D$, each extending the graph L by one of the following cases, where applicable:

1. by an edge of type e_{TG} from v to v if the source and target of e_{TG} are identical,
2. by an edge of type e_{TG} from v to a new and appropriately typed node v',
3. by an edge of type e_{TG} from a new and appropriately typed node v' to v;

furthermore, for each node v^* in L that may be connected to v via an edge of type e_{TG}, create an application condition forbidding the existence of such an edge. Then, $\text{Appl}(b)$ is the conjunction of those application conditions.

Example 2.31 (rule applicability condition)**.** Figure 2.18(a) shows a hpyothetical rule $b = \langle (L \hookleftarrow K \hookrightarrow R), \text{true}, \text{true}\rangle$ that can be applied to two subsequent tracks and deletes the first track. The resulting rule applicability condition of b is $\text{Appl}(b) = \bigwedge_{1 \le i \le 6} \neg\exists a_i$. It forbids the existence of any additional edge adjacent to the deleted track t1. Here, these may be of type next, connecting t1 with other tracks (or, theoretically, itself) or of type isAt, if there is a shuttle located on t1. Note that the application condition do not only need to consider the existence of other nodes with connections to t1, as in $\neg\exists a_3$, $\neg\exists a_5$, and $\neg\exists a_6$, but also potential further connections to nodes in L ($\neg\exists a_1$, $\neg\exists a_2$, and $\neg\exists a_4$). △



(a) Example graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), \text{true}, \text{true} \rangle$

(b) Application condition $\text{Appl}(b) = \bigwedge_{1 \le i \le 6} \neg\exists a_i$ over L

Figure 2.18. – Example rule rule b and rule applicability condition $\text{Appl}(b)$

We also need a mechanism to transfer application conditions to new context while keeping a condition's meaning intact. In particular, we can equivalently transfer application conditions over morphisms by the Shift-construction [EGH+14]. To focus on the requirements of our approach, we slightly adapt the construction: it concerns application conditions consisting only of injective morphisms in subconditions (i.e. conditions in injective normal form), conditions are transferred over injective morphisms only, and only satisfiability by injective morphisms is taken into account. In the literature, the A-transformation [Pen09] is a special case of the Shift-construction [EGH+14] and focuses on the application scenario described; however, we use a modified form of the Shift-construction:

Lemma 2.32 (Shift-lemma, modified [EGH+14])**.** *There is a construction* Shift *such that, for every injective morphism $b : P \hookrightarrow P'$ and every application condition ac over P in injective normal form,* $\text{Shift}(b, ac)$ *transfers ac over b into an application condition over P' in injective normal form such that for every injective morphim $g : P' \hookrightarrow G$ we have $g \circ b \vDash ac \Leftrightarrow g \vDash$* $\text{Shift}(b, ac)$*.*

$$\begin{tikzcd} ac \triangleright P \arrow[rr, hook, "b"] \arrow[dr, hook, "g \circ b \vDash ac"'] & {} \arrow[d, phantom, "="] & P' \triangleleft \text{Shift}(b,ac) \arrow[dl, hook', "g \vDash \text{Shift}(b,ac)"] \\ & G & \end{tikzcd}$$

Construction (Shift-construction, modified [EGH+14])**.** *The Shift-construction is inductively defined as follows:*

$$\begin{tikzcd} P \arrow[r, hook, "b"'] \arrow[d, hook, "a"] \arrow[dr, phantom, "="] & P' \arrow[d, hook', "a'"'] \\ ac \triangleright C \arrow[r, hook, "b'"] & C' \end{tikzcd}$$

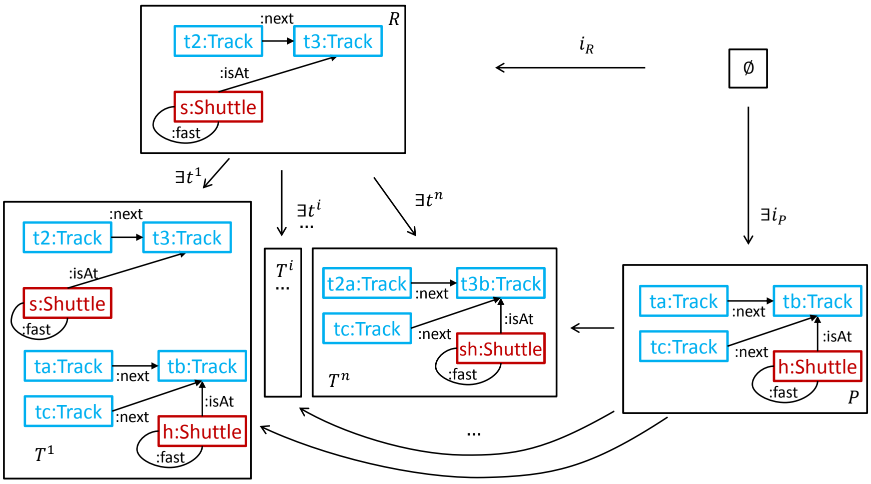


Figure 2.19. – Example application of Shift-construction with $\mathrm{Shift}(i_R, \exists i_P) = \bigvee_{i \in I} \exists t^i$

1. $\mathrm{Shift}(b, \mathrm{true}) = \mathrm{true}$,
2. $\mathrm{Shift}(b, \exists(a, ac)) = \bigvee_{(a',b') \in \mathcal{F}} \exists(a', \mathrm{Shift}(b', ac))$ *with* $\mathcal{F}$ *the set of injective and jointly surjective morphism pairs* (a', b') *such that* $b' \circ a = a' \circ b$,
3. $\mathrm{Shift}(b, \neg ac) = \neg\,\mathrm{Shift}(b, ac)$, *and*
4. $\mathrm{Shift}(b, \bigwedge_{i \in I} ac_i) = \bigwedge_{i \in I} \mathrm{Shift}(b, ac_i)$.

Proof. The proof for the Shift-construction [EGH+14] can be modified to fit the modifications noted above [Dyc12]. □

The construction handles boolean operators in the usual fashion; the only part more involved is the transformation of conditions of the form $\exists(a, ac)$. Given a morphism $g' : P \hookrightarrow G$, satisfaction of $\exists(a, ac)$ over P describes a possible extension of P (in G) to C (such that ac is satisfied). However, when the condition is transferred to the context of P' via b, P' may already have the extension to C – or parts of it – present. Hence, there may now be multiple possibilities to complete the extension. These options are described by the disjunction $\bigvee_{(a',b') \in \mathcal{F}} \exists(a', \mathrm{Shift}(b', ac))$. Intuitively, we build overlappings of C and P', then, in each overlapping, look for the parts still missing to have the extension to C.

Example 2.33 (Shift-lemma)**.** In our approach we will frequently apply the Shift-construction to transfer graph constraints to the context of the (symbolic) result of a rule application, i.e. the right side of a graph rule. Since graph constraints are application conditions over the empty graph, they can be transferred similar to regular application conditions. In particular, consider Figure 2.19: we shift the violation of our safety property – the existence of a fast shuttle on a switch – to the right rule side R of f2f via i_R. The result is a disjunction of conditions $\exists t^i$ over R describing situations containing the right rule side and the violation specified by P. Then, by the Shift-lemma, any graph G with a morphism $g : R \hookrightarrow G$ and $g \models \bigvee_{i \in I} \exists t^i$ will also satisfy $\exists i_P$, i.e. violate the safety property. Conversely, $G \models \exists i_P$ implies $g \models \bigvee_{i \in I} \exists t^i$. △

Application conditions can also be transferred over graph rules. Given an application condition ac over the right side of a graph rule, we can compute an application condition ac' over the left rule side such that, for each application of the rule, satisfiability of ac' by the match is equivalent to satisfiability of ac by the comatch. This construction is the reason why graph rules with left application conditions only are equally expressive to graph rules with both left and right application conditions [HP09]. As with the Shift-construction, we will include a minor modification and require condition in injective normal form.

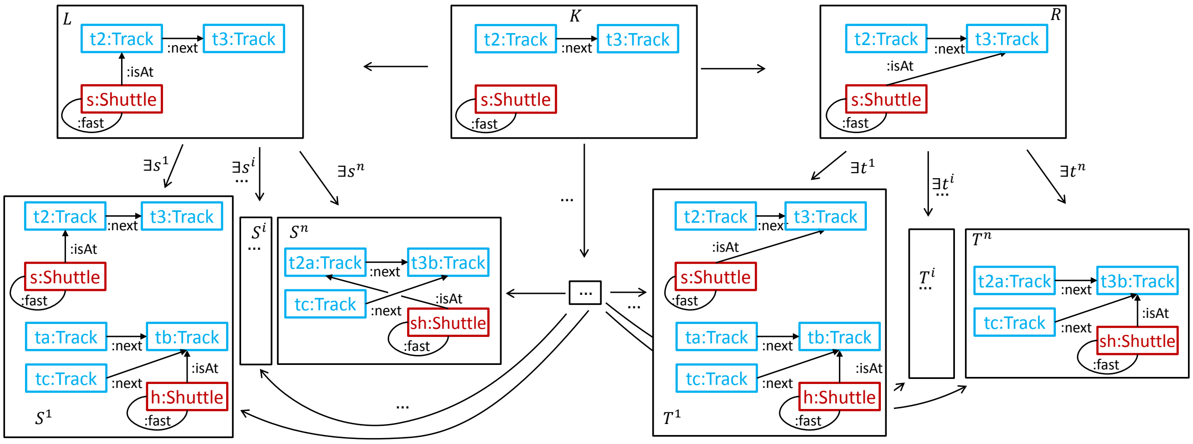


Figure 2.20. – Example application of L-construction with $\mathrm{L}(\mathsf{f2f}, \bigvee_{i\in I} \exists t^i) = \bigvee_{i\in I} \exists s^i$

Lemma 2.34 (L-lemma [HP09, EGH⁺14])**.** *There is a construction* L *such that for every rule* $b = \langle L \hookleftarrow K \hookrightarrow R\rangle$ *and every application condition* ac *over* R *and in injective normal form,* L *transforms* ac *via* b *into an application condition over* L *in injective normal form such that for every transformation* $G \Rightarrow_{b,m,m'} H$*, we have* $m \vDash \mathrm{L}(b, ac) \Leftrightarrow m' \vDash ac$*.*

Construction (L-construction, modified [HP09, EGH⁺14])**.** *The L-construction is inductively defined:*

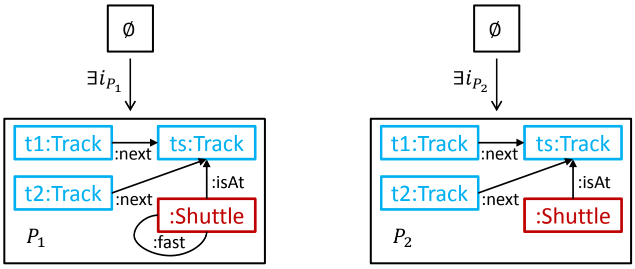
$$\begin{tikzcd}
L \arrow[d, hook, "a'"] \arrow[dr, phantom, "(2)"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"'] \arrow[d, hook] \arrow[dr, phantom, "(1)"] & R \arrow[d, hook', "a"'] \\
\mathrm{L}(b',ac) \triangleright L' & K' \arrow[l, hook', "l'"] \arrow[r, hook, "r'"] & R' \triangleleft ac
\end{tikzcd}$$

1. $\mathrm{L}(b, \mathrm{true}) = \mathrm{true}$,
2. $\mathrm{L}(b, \exists(a, ac)) = \exists(a', \mathrm{L}(b', ac))$ *if* $b' = \langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ *constructed via the pushouts (1) and (2) exists and false, otherwise,*
3. $\mathrm{L}(b, \neg ac) = \neg\mathrm{L}(b, ac)$*, and*
4. $\mathrm{L}(b, \bigwedge_{i\in I} ac_i) = \bigwedge_{i\in I} \mathrm{L}(b, ac_i)$*.*

Proof. We refer to the respective source [HP09]; injectivity of a' in $\exists a' = \mathrm{L}(b, \exists a)$ in case (2) follows from the properties of the pushouts (1) and (2) (Fact 2.18 (p. 29)). □

Example 2.35 (L-lemma)**.** In our algorithms, we will frequently transform application conditions defined over a rule's right side to the rule's left side to compute the symbolic state before application of the rule in question. Consider $ac = \bigvee_{i\in I} \exists t^i$ from Example 2.33 (p. 40). Figure 2.20 shows its transformation over the f2f rule; in particular, $ac' = \mathrm{L}(\mathsf{f2f}, \bigvee_{i\in I} \exists t^i) = \bigvee_{i\in I} \exists s^i$. Intuitively, each situation described by the disjunction's operands $\exists t^i$ has had the rule reversely applied to it. The result is a disjunction of situations that, after rule application, will lead to the corresponding situations described by the condition(s) over R. Then, for each transformation $G \Rightarrow_{\mathsf{f2f},m,m'} H$, we have the equivalence $m \vDash ac' \Leftrightarrow m' \vDash ac$. △

We then need mechanisms by which graph constraints can be compared. One obvious aspect is the equivalence of graph constraints, which is based on the equivalence of application conditions. A second aspect is implication of graph constraints. A graph constraint C_1 implies a second graph constraint C_2, if all graphs satisfying C_1 also satisfy C_2. Intuitively, the set of graphs represented by the implying constraint C_1 is a subset of the graphs encoded by the



(a) $C_1 = \exists i_{P_1}$

(b) $C_2 = \exists i_{P_2}$

Figure 2.21. – Graph constraints $C_1 = \exists i_{P_1}$ and $C_2 = \exists i_{P_2}$ with $C_1 \vDash \mathcal{C}_2$

implied graph constraint C_2. The concept and problem of implication has been formalized and thoroughly discussed on a categorical level [Pen09]; here, we focus on its interpretation for the category of typed graphs.

Definition 2.36 (implication of graph constraints [Pen09])**.** *Given two graph constraints $\mathcal{C}_1$ and $\mathcal{C}_2$, we say that $\mathcal{C}_1$* implies *$\mathcal{C}_2$, denoted as $\mathcal{C}_1 \vDash \mathcal{C}_2$, if, for all graphs G, it holds that $G \vDash \mathcal{C}_1$ implies $G \vDash \mathcal{C}_2$.*

Example 2.37 (implication of graph constraints)**.** Consider the graph constraints $C_1 = \exists i_{P_1}$ and $\mathcal{C}_2 = \exists i_{P_2}$ shown in Figures 2.21(a) and 2.21(b), respectively. C_1 is satisfied by all graphs containing the subgraph P_1 – a shuttle, located on a switch, driving fast. C_2 is satisfied by graphs containing P_2 – a shuttle that is located on a switch. It is easy to see that all graphs satisfying C_1 will always satisfy C_2: P_2 is a subgraph of P_1 and hence, a graph that contains P_1 (i.e. fulfills C_1) will always contain P_2 – and hence, fulfill C_2.

The reverse, however, is obviously not true: the graph P_2 trivially satisfies C_2 but does not satisfy C_1 because it is missing the fast edge. Thus, C_2 does not imply C_1. Besides P_2, we can think of other graphs satisfying C_2 while not satisfying C_1 – we can just extend P_2 by nodes and edges that do not lead to an occurrence of P_1.

If we consider constraints $\neg C_1 = \neg\exists i_{P_1}$ and $\neg C_2 = \neg\exists i_{P_2}$, we can see that the latter implies the former. Both constraints are satisfied by graphs where the respective subgraph is absent. Then, any graph that does not have P_2 as a subgraph (i.e. satisfies C_2) cannot have that graph plus the fast edge – P_1 – as a subgraph. The reverse, $\neg C_1 \vDash \neg C_2$, is not true. △

Note that the constraints in the example above are rather simple in their structure: they are singular (negated) existential conditions without further levels of nesting. In general, given complex constraints in addition to an infinite number of graphs satisfying a given graph constraint, we cannot perform analysis of implication by hand. Furthermore, Definition 2.36 does not provide a constructive approach to answering the question of implication. The problem can be reduced to satisfiability of graph constraints: if and only if C_1 implies C_2, $\neg C_1 \vee C_2$ (equivalent to the direct implication $C_1 \Rightarrow C_2$) can be satisfied by at least one graph [HP09, Pen09]. We know that satisfiability of graph constraints is undecidable in general [HP09]; then, so is the question of implication of graph constraints. Approaches and algorithms to solve the question of implication of graph constraints have been established for the general case in the literature [Pen09, SLO17, SLO18].

Finally, we can establish a connection between graph constraints and application conditions that are not graph constraints. In particular, we want to transform application conditions into graph constraints in order to compare the result to other graph constraints. As with the Shift- and L-constructions, this procedure has been described for more general categories [HP09]; here, we will use a version applicable to the category of typed graphs:

Lemma 2.38 (reduction to a graph constraint)**.** *There is a construction such that for each application condition ac over a graph A in injective normal form, the* reduction of ac to a graph constraint*, denoted $ac_{|\varnothing}$, is a graph constraint in injective normal form and we have: for each graph G, there is an injective morphism $g : A \hookrightarrow G$ with $g \vDash ac$ if and only if $G \vDash ac_{|\varnothing}$.*

Construction. *Reduction to a graph constraint is inductively defined as follow:*

$$\begin{tikzcd}
A \arrow[d, hook, "a"] & \varnothing \arrow[dl, hook, "i_B"] \\
ac \triangleright B &
\end{tikzcd}$$

1. $\text{true}_{|\varnothing} = \exists i_A$ *(where* true *is an application condition over A),*
2. $(\exists(a : A \hookrightarrow B, ac))_{|\varnothing} = \exists(i_B, ac)$*,*
3. $(\neg ac)_{|\varnothing} = \neg ac_{|\varnothing}$*, and*
4. $(\bigwedge_{i\in I} ac_i)_{|\varnothing} = \bigwedge_{i\in I} ac_{i|\varnothing}$*.*

Proof. We will prove the equivalence for cases (1) and (2); these can then be extended over the boolean combinations in (3) and (4).

1. Consider ac = true as an application condition over A. Then, $ac_{|\varnothing} = \exists i_A$ and $G \vDash \exists i_A$ is equivalent to the existence of an injective morphism $g : A \hookrightarrow G$.
2. Consider $ac = \exists(a, ac_B)$ and $ac_{|\varnothing} = \exists(i_B, ac_B)$ as shown below. We will prove both directions separately.

$$\begin{tikzcd}
A \arrow[d, hook, "a"] \arrow[dd, hook, bend right, "g"'] & \varnothing \arrow[dl, hook, "i_B"] \arrow[ddl, hook, bend left, "i_G"] \\
ac_B \triangleright B \arrow[d, hook, "q"] & \\
G &
\end{tikzcd}$$

If. Consider a graph G with $G \vDash ac_{|\varnothing}$. Then, there exists an injective morphism $q : B \hookrightarrow G$ such that $q \circ i_B = i_G$ and $q \vDash ac_B$. Thus, there is an injective morphism $g : A \hookrightarrow G$ with $g = q \circ a$ and hence, $g \vDash \exists(a, ac_B)$.
Only if. Consider a graph G with an injective morphism $g : A \hookrightarrow G$ such that $g \vDash ac$. Then, there is an injective morphism $q : B \hookrightarrow G$ such that $q \circ a = g$ and $q \vDash ac_B$. With $q \circ i_B = i_G$, we have $G \vDash \exists(i_B, ac_B)$.

□

Example 2.39 (reduction to a graph constraint)**.** Consider $ac = \bigvee_{i\in I} \exists s^i$ from Example 2.35 (p. 41), shown again in Figure 2.22. The reduction to a constraint $ac_{|\varnothing} = \bigvee_{i\in I} \exists i_{S^i}$ of ac is also shown in Figure 2.22. Then, any graph G with an injective morphism $g : L \hookrightarrow G$ and $G \vDash ac$ implies $G \vDash ac_{|\varnothing}$; conversely, satisfaction of $ac_{|\varnothing}$ by a graph G implies the existence of an injective morphism $g : L \hookrightarrow G$ such that $g \vDash ac$. △

This concludes the set of constructions and transformations for application conditions and graph constraints relevant for our approach to inductive invariant checking.

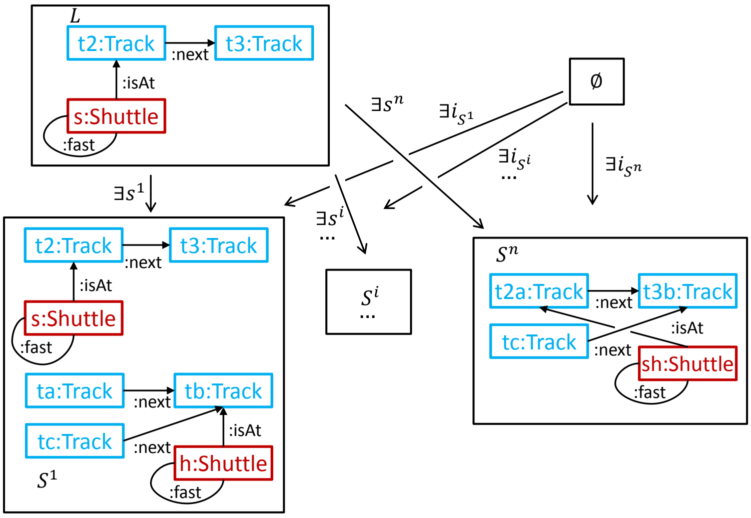


Figure 2.22. – Reduction $ac_{|\varnothing} = \bigvee_{i\in I} \exists i_{S^i}$ of $ac = \bigvee_{i\in I} \exists s^i$

3. Prerequisites: Formalizing 1-Inductive Invariant Checking

Given the formal foundations (re-)introduced in Chapter 2, we can now focus on formalizing our verification approach. This chapter will transfer the informal description of 1-inductive invariant checking in Section 1.2 to the formal level and, in doing so, reiterate the current situation of 1-inductive invariant checking for graph transformation systems described in earlier work [BBG+06, Dyc12]. Extending this formalization in different directions, with k-inductive invariant checking one of those directions, will then be the focus of Chapter 4.

While we will discuss inductive invariant checking as a concrete verification approach in this chapter and the next, the formalization will remain abstract in the sense that the approaches cannot be directly implemented yet. They will provide a number of proof obligations that solve the corresponding verification problems. However, they will not contain constructions or algorithms that can be implemented as executable code – this will be addressed in Chapter 5 for the general approach (**Formal-general**) and in Chapter 6 for the restricted approach (**Formal-restricted** and **Impl.-restricted**) to k-inductive invariant checking. As a result, this chapter will only discuss 1-invariant checking at the level of the basic formal model introduced in Chapter 2. Since 1-inductive invariant checking for graph transformation systems has been established, implemented, and applied for a limited version of this basic formal model in earlier work [BBG+06, BG08b, BG08a, Dyc12], we will not reiterate its implementation and algorithmic perspective here or elsewhere in this thesis. Likewise, the basic formal model used in this chapter will not reflect restrictions of those earlier implementations – they will be compared with the results of this thesis in the evaluation (Chapter 9) instead.

Outline. Figure 3.1 attempts to provide an overview of this chapter's structure in terms of formal elements. We focus on Verification Problem 3.1, which is concerned with establishing a graph constraint's (safety property's) validity in a singular graph grammar's state space. Lemma 3.4 will describe proof obligations of the existing verification approach [BBG+06, Dyc12] of 1-inductive invariant checking: if the graph constraint is an 1-inductive invariant (Definition 3.3) of the graph grammar's graph transformation system – i.e. is preserved by all rule applications – and if this inductive step is combined with the constraint's validity in the grammar's start graph as the base case, the constraint is satisfied in all graphs of the grammar's state space.

As our starting point, we recall the generic verification problem established in Chapter 1:

Verification Problem 1.1. *Given a system that is defined by a metamodel, an initial system state, and specification of system behavior and given a set of safety properties, does every state in the system's state space satisfy the safety properties?*

In general terms, we want to establish a system's correctness by proving that the system in question cannot provoke a violation of a safety property. We recall the mapping of system elements to formal concepts by the basic formal model (cf. Chapter 2):

Formal Model (basic, implicitly used in earlier work [BBG+06, Dyc12])**.** *Systems and system specifications consist of the following elements:*

System metamodels *are specified by type graphs (Definition 2.8 (p. 21)).*

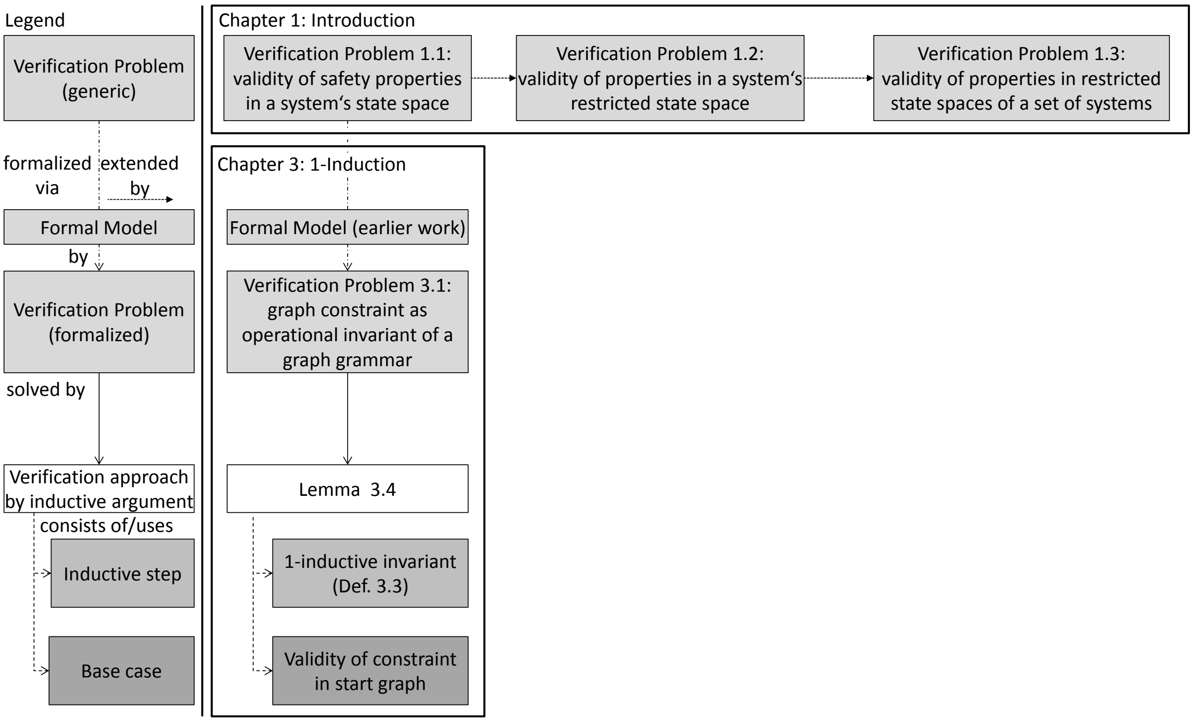

Legend
Verification Problem (generic)
formalized via
extended by
Formal Model
by
Verification Problem (formalized)
solved by
Verification approach by inductive argument
consists of/uses
Inductive step
Base case
Chapter 1: Introduction
Verification Problem 1.1: validity of safety properties in a system's state space
Verification Problem 1.2: validity of properties in a system's restricted state space
Verification Problem 1.3: validity of properties in restricted state spaces of a set of systems
Chapter 3: 1-Induction
Formal Model (earlier work)
Verification Problem 3.1: graph constraint as operational invariant of a graph grammar
Lemma 3.4
1-inductive invariant (Def. 3.3)
Validity of constraint in start graph

Figure 3.1. – Overview and dependencies of definitions, verification problems, and lemmas

System states *– including initial states – are described by typed graphs (Definitions 2.1 (p. 20) and 2.8 (p. 21)).*

System behavior *is described by a typed graph transformation system (Definition 2.24 (p. 34)), which consists of typed graph transformation rules (Definition 2.20 (p. 32)).*

Properties *are modeled as graph constraints (Definition 2.12 (p. 25)).*

Systems *are specified by typed graph grammars (Definition 2.28 (p. 36)), which consist of an initial state – a start graph – and a typed graph transformation system.*

System state spaces *are described by the state spaces (Definition 2.28 (p. 36)) – the set of all reachable graphs – of the corresponding graph grammars.*

Given these system elements, we formalize the validity of a graph constraint in a graph grammar's state space – the notion of operational invariants (Definition 1.9 (p. 10)) – as follows:

Definition 3.1 (operational invariant)**.** *Given a graph grammar GG and a graph constraint $\mathcal{F}$, we say that $\mathcal{F}$ is an* operational invariant *of GG, if, for all graphs $G \in \mathrm{REACH}(GG)$, we have $G \models \mathcal{F}$.*

With that, we can rephrase our verification question in formal terms as follows:

Verification Problem 3.1. *Given a graph grammar of the form $GG = (GTS, G_0)$ with a start graph G_0 and a graph transformation system GTS and given a graph constraint $\mathcal{F}$, is $\mathcal{F}$ an operational invariant of GG?*

This verification problem is concerned with a graph constraint instead of a set of properties (as in Verification Problem 1.1). However, since a graph constraint can contain subconditions in boolean combinations, we can express a set of properties by using a graph constraint that conjunctively joins the graph constraints corresponding to the properties.

Example 3.2 (shuttle system)**.** We will use the shuttle system introduced in Chapters 1 and 2 as our running example. The system is modeled as:

– a *graph grammar* $GG = (GTS, G_0)$ with
– a *start graph* G_0 (Figure 3.2(i)), which specifies the system's initial state and
– a *graph transformation system* $GTS = (TG, \mathcal{R})$, which describes system behavior and consists of
– a *type graph* TG (Figure 3.2(a)), which specifies the system metamodel and
– a set of graph rules $\mathcal{R}$ (Figures 3.2(b)–3.2(h)).
– Furthermore, a *graph constraint* $\mathcal{F}$ (Figure 3.3) specifies the safety property, which, as formalized by Verification Problem 3.1, should be satisfied
– in the graph grammar's *state space* $\text{REACH}(GG)$.

As before, the system's behavior describes the protocol governing a single shuttle's speed mode changes. A shuttle may always choose to slow down or stay in speed mode slow; increasing its velocity towards acc or fast, on the other hand, is bound to the absence of switches ahead of the shuttle.

Note that G_0 is not depicted in its entirety: a number of tracks to the right have been left out. However, the part not depicted does not contain any additional shuttles. We could have used a smaller start graph, but that does not properly illustrate the problems to be addressed in our verification approach.

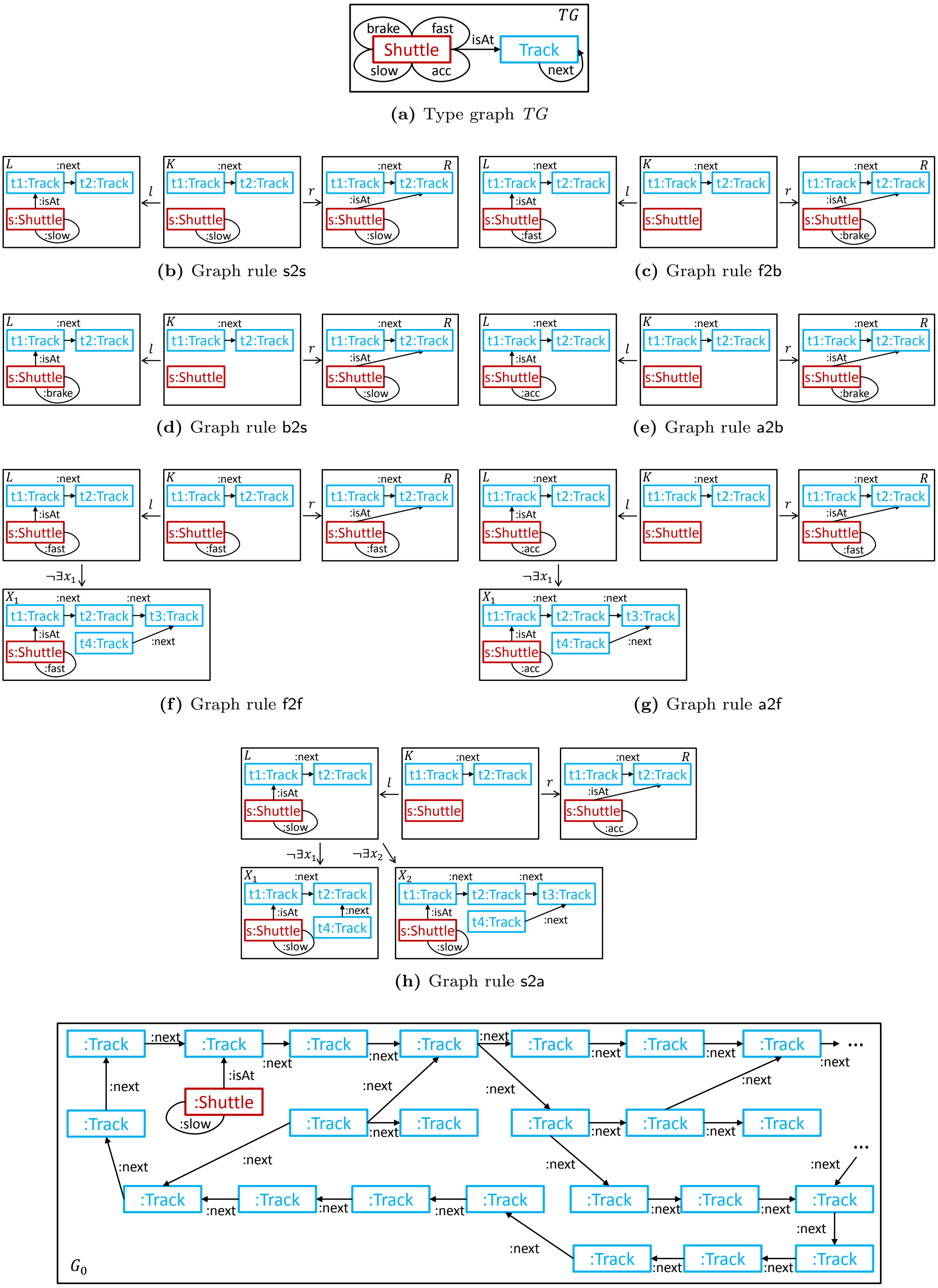
We want to prevent a shuttle from reaching a switch while in speed modes fast, acc, or brake – whenever a shuttle drives on a switch, it should do so in speed mode slow. △

A positive answer to Verification Problem 3.1 (p. 46) above means that every system state satisfies the safety properties in $\mathcal{F}$ and that the system is considered safe. As has been discussed in Chapter 1, a common and well-researched approach to solve this and similar verification problems is model checking [BK08, CHVB18]. In particular, model checking has seen widespread application in the verification of graph transformation systems [Ren04, GdMR+12, ABJ+10]. In Algorithm 3.1, we sketch a simple explicit-state model checking algorithm applicable to a graph grammar to demonstrate the general idea.

Algorithm 3.1: Simple model checking algorithm

```
input  : a graph grammar $GG = ((TG, \mathcal{R}), G_0)$ and a graph constraint $\mathcal{F}$
output: a graph $G \in \text{REACH}(GG)$ such that $G \not\models \mathcal{F}$ or null if no such graph exists
states ← {$G_0$}
while states.isNotEmpty() do
    state ← states.removeFirst()
    foreach $b \in \mathcal{R}$ do
        foreach match of $b$ in state do
            $G$ ← applyRule(state, $b$, match)
            if $G \not\models \mathcal{F}$ then
                return $G$
            state.add($G$)
return null
```

As argued in Section 1.1, we can see that infinite systems pose a serious problem for approaches following this scheme. Infinite applicability of graph rules may still lead to a finite state space if there are cycles in the state space and if newly created states are checked for



(a) Type graph TG

(b) Graph rule s2s

(c) Graph rule f2b

(d) Graph rule b2s

(e) Graph rule a2b

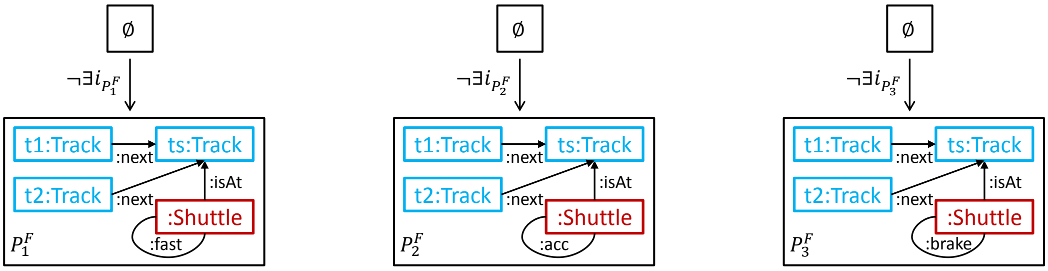
(f) Graph rule f2f

(g) Graph rule a2f

(h) Graph rule s2a

(i) Start graph G_0

Figure 3.2. – Graph grammar $GG = (GTS, G_0)$ with $GTS = (TG, \mathcal{R})$



(a) Graph constraint $\neg F_1 = \neg\exists i_{P_1^F}$ **(b)** Graph constraint $\neg F_2 = \neg\exists i_{P_2^F}$ **(c)** Graph constraint $\neg F_3 = \neg\exists i_{P_3^F}$

Figure 3.3. – Safefy property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$

isomorphisms to states already considered. However, state spaces with an infinite number of distinct graphs will lead to a non-terminating algorithm.

Furthermore, a verification result gained from application of the above scheme applies only to the graph grammar in question, which consists of a specific start graph and graph transformation system. Changes to the start graph and the graph transformation system will invalidate the verification result and require costly repetitions of the verification scheme.

Even with a fixed graph transformation system, explicit-state model checking of graph grammars with a set of possible initial states creates challenges: all state spaces for all initial states would have to be calculated and analyzed for the respective constraint's validity. If the number of allowed initial states is infinite, this is not possible without considering symbolic approaches or under-approximation. Even when the number of initial states is finite, the increase in computational effort seems at odds with the fact that the abstract system behavior – described by graph transformation rules – does not change at all. For both reasons, it can be desirable to split the verification scheme into two distinct tasks: reasoning about a graph grammar's initial state and reasoning about system behavior (in the form of graph rules). In order to achieve the latter, i.e. to analyze system behavior without considering initial states, we have to abstract from a concrete state space; in other words, we have to apply a symbolic approach.

We have discussed 1-inductive invariant checking [BBG$^+$06, BG08b, Dyc12] as one such approach in Section 1.2. Although previous work has used the term inductive invariant checking, we will refer to it here as 1-inductive invariant checking in order to avoid confusion with k-inductive invariant checking. By Definition 1.8 (p. 10), a 1-inductive invariant is a property (here: a graph constraint) whose validity is preserved by execution of a single step of system behavior (here: the application of a graph rule). In terms of our basic formal model, we define:

Definition 3.3 (1-inductive invariant)**.** *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and a graph constraint $\mathcal{F}$, $\mathcal{F}$ is an* 1-inductive invariant *of GTS, if, for all rules $b \in \mathcal{R}$, the following condition holds:*

$$\forall G, G' ((G \Rightarrow_b G') \Rightarrow (G \vDash \mathcal{F} \Rightarrow G' \vDash \mathcal{F}))$$

We also say that GTS preserves *$\mathcal{F}$.*

In other words, a 1-inductive invariant is a graph constraint whose validity in a graph (here: $G \vDash \mathcal{F}$) before the application of a graph rule ($G \Rightarrow_b G'$) implies its validity after rule application ($G' \vDash \mathcal{F}$). Since all transformations $G \Rightarrow_b G'$ are considered, this property relates to the nature of the graph transformation rules in question, not to specific graphs before or after rule application. Thus, the definition is independent from a graph grammar's state space or initial state. Then, 1-inductive invariants can be used as part of an inductive argument: if the graph constraint in question is always preserved by all possible graph rule applications,

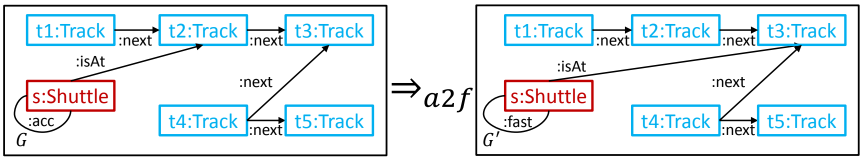


Figure 3.4. – Example transformation $G \Rightarrow_{\mathsf{a2f}} G'$ with $G \vDash \mathcal{F}$ and $G' \nvDash \mathcal{F}$

and if it is satisfied in a graph grammar's initial state, it holds in all states of the state space – it is an operational invariant:

Lemma 3.4 (operational invariants of graph grammars)**.** *Let $GG = (GTS, G_0)$ be a graph grammar with a graph transformation system $GTS = (TG, \mathcal{R})$ and let $\mathcal{F}$ be a graph constraint. $\mathcal{F}$ is an operational invariant of GG, if the following conditions hold:*

1. *$G_0 \vDash \mathcal{F}$.*
2. *$\mathcal{F}$ is a 1-inductive invariant of GTS.*

Proof. Consider a graph G in the graph grammar's state space, i.e. $G \in \mathrm{REACH}(GG)$. Hence, there exists a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$. We will prove $G \vDash \mathcal{F}$ by induction over n:

Base case. For $n = 0$, we have $G = G_0$ and, by precondition, $G \vDash \mathcal{F}$.

Inductive step. Consider the sequence $G_0 \Rightarrow^{n+1}_{\mathcal{R}} G$ with $n + 1 > 0$. Then, there is a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$. By inductive hypothesis, we have $G' \vDash \mathcal{F}$. By definition of 1-inductive invariants (Definition 3.3), the existence of the transformation $G' \Rightarrow_{\mathcal{R}} G$, and $G' \vDash \mathcal{F}$ and since $\mathcal{F}$ is a 1-inductive invariant, we have $G \vDash \mathcal{F}$, concluding the inductive proof. □

Lemma 3.4 describes a verification technique that solves Verification Problem 3.1 (p. 46). Furthermore, as required, it introduces separately verifiable conditions for the graph grammar's start graph and its graph transformation system. By formulating the verification question independent of a specific start graph, we decouple the initial system state and system behavior. As a consequence, our verification approach requires safety issues to be addressed and ensured at the behavioral level (i.e., at the level of graph rules) and not at the level of individual (initial) system states. This is especially useful when the same system behavior is used in multiple and slightly differing instances. For example, once successfully verified, the behavioral part of the shuttle protocol can be safely used in different topologies and start graphs.

While Lemma 3.4 does not explicitly compute the graph grammar's state space as part of the verification process, the definition of 1-inductive invariants still reasons about all possible transformations between graphs (cf. $\forall G, G'(G \Rightarrow_{\mathcal{R}} G' ...)$). Explicit analysis of all those transformations, however, is similarly infeasible to the exhaustive computation and analysis of the system's state space. Hence, we require a symbolic technique to establish a graph constraint as a 1-inductive invariant. Earlier work [BBG⁺06, Dyc12] describes our solution for a restricted formal model. In this thesis, Chapters 5 and 6 will do so for k-inductive invariant checking with respect to a general (Chapter 5) and restricted formal model (Chapter 6).

Example 3.5. Recall the graph grammar and safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ from Example 3.2, Figure 3.3 (p. 49). When applying Lemma 3.4 to this example, it is quite easy to verify the first condition $G_0 \vDash \mathcal{F}$: since G_0 does not contain a fast, accelerating, or braking shuttle (on a switch or elsewhere) as a subgraph, the initial state satisfies the safety property. If the absence of this situation is preserved by application of graph rules, i.e. if $\mathcal{F}$ is a 1-inductive invariant of GTS, all graphs reachable via application of graph rules from G_0 will satisfy $\mathcal{F}$.

In this case, however, we have already discussed in Example 1.13 (p. 14) that $\mathcal{F}$ will not be a 1-inductive invariant of our graph transformation system. Figure 3.4 shows a graph transformation $G \Rightarrow_{\mathsf{a2f}} G'$ where the origin of the transformation satisfies $\mathcal{F}$ while the result graph G' does not – it violates $\neg F_1$. Note that neither G nor G' actually appear in REACH(GG): the analysis of graph rules is independent from the initial graph and state space. However, since $\mathcal{F}$ is not a 1-inductive invariant, we cannot use the inductive argument of Lemma 3.4 to establish that the system is safe.

In order to have $\mathcal{F}$ as a 1-inductive invariant, we may attempt to further modify our graph transformation rules, i.e. system behavior, by adding more application conditions that prevent a shuttle from accelerating before arriving at a switch. However, proving that $\mathcal{F}$ is a 1-inductive invariant is non-trivial even for this small example – which is why previous work has formalized and implemented a symbolic verification algorithm [BBG+06, Dyc12].

Since G_0 and the state space are finite, we could also employ explicit-state model checking for both examples. However, we would have to repeat the verification whenever the track topology changes – and, for a very large start graph G_0, 1-inductive invariant checking may be more efficient. Furthermore, if we allowed rules adding tracks to our system, we would have an infinite state space. △

Conclusion. This chapter has reiterated the existing formal approach to 1-inductive invariant checking for graph transformation systems [BBG+06, Dyc12]. With respect to Verification Problem 1.3 (p. 13) and the challenges of symbolic approaches in general and 1-inductive invariant checking in particular, this chapter leaves us with three important issues:

State space restrictions. In some cases, we want to impose restrictions on state spaces of systems to be verified. In particular, we may want to consider only states that are reachable via paths whose graphs fulfill a graph constraint. There are several reasons for this approach: the constraint may reflect assumptions about the systems out of scope of the verification. The constraint may also reflect proven guarantees about the systems, which can be used to support the verification. Restrictions of state spaces are included in Verification Problem 1.2 (p. 13) and its successor, Verification Problem 1.3 (p. 13).

Infinite number of start graphs. Lemma 3.4 (p. 50) and Verification Problem 1.1 (p. 1) refer to one graph grammar with a specific start graph. While the verification of $G_0 \vDash \mathcal{F}$ for a start graph G_0 and graph constraint $\mathcal{F}$ is not computationally challenging, it needs to be repeated with every change in the start graph. In order to solve Verification Problem 1.3 (p. 13) and truly achieve independence from specific start graphs, we would have to consider all possible start graphs. Given the current formalism, that is either infeasible – because the number of start graphs might be infinite – or unnecessarily restrictive – because we would need to explicitly supply a finite set of possible start graphs.

Completeness and false negatives. The technique's inductive step in Lemma 3.4 (p. 50) is one of the main strengths of the approach: it only refers to a (fixed) set of graph transformation rules, not (initial) system states. However, that strength comes with a specific flaw: locality and lack of context information. While our approach and implementation of 1-inductive invariant checking is sound in the sense that it will never erroneously report graph constraints to be 1-inductive invariants, counterexamples can be false negatives [Dyc12]. We wish to reduce that number of false negatives (and, in doing so, address **Appl.-deg.completeness**). Therefore, this thesis and its preceding work *[3, 4]* extend 1-inductive invariants to k-inductive invariants. Taking not only singular transformations but a path of transformations into account accumulates more information about the situation and system in question and reduces the number of false negatives.

4. From 1-Induction to k-Induction

In order to address some of the problems and limitations of 1-inductive invariant checking, this thesis aims at extending the approach in several directions, most notably towards k-induction. This chapter will formalize the informal approach to k-inductive invariant checking described in Section 1.3, similar to Chapter 3 and the corresponding Section 1.2 for 1-inductive invariant checking. As such, this chapter is the first step towards **Formal-general**, **Formal-restricted**, and **Impl.-restricted**, which make up the main contribution of this thesis. However, we will still remain on an abstract level in this chapter. Similar to Chapter 3, which discussed 1-inductive invariant checking on the basis of a basic formal model, this chapter will address its extension on the basis of what will be called the general formal model. Constructions and an algorithmic view will then appear in Chapters 5 for said general formal model – and, alongside the implementation, in Chapter 6 for a restricted formal model.

Outline. Figure 4.1 provides an overview of this chapter's structure. In Chapter 3, we recalled the definition for 1-inductive invariants. Inductive invariants were used in Lemma 3.4 in order to solve Verification Problem 3.1, which is concerned with establishing a safety property's validity in a graph grammar's state space. In this chapter, we address the issues brought up in Chapter 1 and revisited in Chapter 3, namely restricting systems' state spaces (Verification Problem 1.2) and reasoning about a set of systems (Verification Problem 1.3).

In Section 4.1, we introduce a way to restrict a graph grammar's state space by an additional graph constraint, which we call a *guaranteed constraint.* A guaranteed constraint forbids the occurence of states violating it (in the state space). This notion is formalized in Definition 4.1, which defines a graph grammar's state space under a (guaranteed) constraint. The idea and justification behind this extension will be explained in detail; in short, it allows for a more direct inclusion of additional information and potential characteristica of systems that would otherwise be difficult to formalize.

Using guaranteed constraints, we then formalize Verification Problem 1.2 in Verification Problem 4.1, which is also an extension of Verification Problem 3.1. In Lemma 4.5, we provide a verification approach that solves this new verificaction problem using the notion of 1-inductive invariants under a guaranteed constraint (Definition 4.4). This lemma is an extension of Lemma 3.4, which did not use a guaranteed constraint.

We also extend our approach to reason not only about a singular graph grammar, but, by taking sets of start graphs into account, about a (possibly infinite) set of induced graph grammars. Induced graph grammars (Definition 4.7) are sets of grammars defined by a graph transformation system and a *start configuration constraint* that describes possible start graphs.

Verification Problem VP.1g then uses induced graph grammars to formalize Verification Problems 1.3 and extend Verification Problem 4.1. It is solved by the verification approach described in Lemma 4.9. Like Lemma 4.5, it uses an inductive argument with 1-inductive invariant checking. However, it also requires implication of the safety property by the start configuration constraint to establish the base case of its inductive argument.

In Section 4.2, we extend the notion of 1-inductive invariants to k-inductive invariants, which are defined in Definition D.1: validity of the respective constraint in a path of transformations of length $k-1$ implies its validity for the resulting state after any subsequent transformation. This notion will first be used by Lemma 4.12 to reason about singular graph grammars (Verification Problem 4.1) with an adjustment to the base of induction: the k-inductive invariant has to be

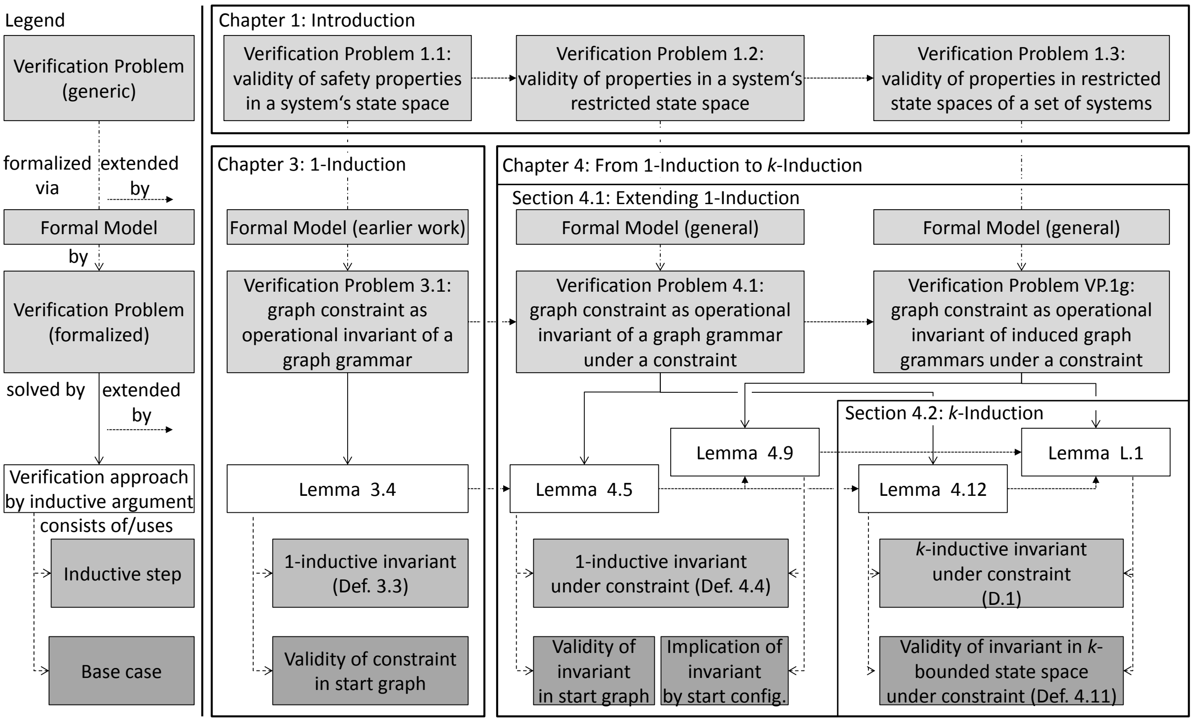

Legend
Verification Problem (generic)
formalized via
extended by
Formal Model
by
Verification Problem (formalized)
solved by
extended by
Verification approach by inductive argument
consists of/uses
Inductive step
Base case
Chapter 1: Introduction
Verification Problem 1.1: validity of safety properties in a system's state space
Verification Problem 1.2: validity of properties in a system's restricted state space
Verification Problem 1.3: validity of properties in restricted state spaces of a set of systems
Chapter 3: 1-Induction
Formal Model (earlier work)
Verification Problem 3.1: graph constraint as operational invariant of a graph grammar
Lemma 3.4
1-inductive invariant (Def. 3.3)
Validity of constraint in start graph
Chapter 4: From 1-Induction to k-Induction
Section 4.1: Extending 1-Induction
Formal Model (general)
Verification Problem 4.1: graph constraint as operational invariant of a graph grammar under a constraint
Lemma 4.5
Lemma 4.9
1-inductive invariant under constraint (Def. 4.4)
Validity of invariant in start graph
Implication of invariant by start config.
Formal Model (general)
Verification Problem VP.1g: graph constraint as operational invariant of induced graph grammars under a constraint
Section 4.2: k-Induction
Lemma 4.12
Lemma L.1
k-inductive invariant under constraint (D.1)
Validity of invariant in k-bounded state space under constraint (Def. 4.11)

Figure 4.1. – Overview and dependencies of definitions, verification problems, and lemmas

valid in all paths of length $k-1$ from the graph grammar's start graph (as opposed to validity in just the start graph). The corresponding formal concept is introduced as k-bounded state space in Definition 4.11.

Our central verification approach, which is formalized in Lemma L.1, will then solve Verification Problem VP.1g. It employs k-inductive invariants (under a guaranteed constraint) to reason about induced graph grammars. As before, the k-inductive invariant will be used as the inductive step of an inductive argument. However, we have to adjust the base of induction again as well: not one $k-1$-bounded state space given a singular start graph has to be considered, but all $k-1$-bounded state spaces from all possible start graphs described by the start configuration constraint.

Lastly, Lemma L.2 describes a verification approach that combines Lemma L.1 with an additional verification step for the guaranteed constraint. In some cases, explicit verification of the guaranteed constraint can be required.

Note the orthogonality of two dimensions here: both 1-induction and k-induction can be used to solve Verification Problem 4.1, which addresses a singular graph grammar, and Verification Problem VP.1g, which addresses a set of induced graph grammars.

4.1. Extending 1-Induction

First, we will address the point of imposing limitations on a system's state space, i.e. limit the states the system can assume (by rule application). We recall the corresponding verification problem from Chapter 1:

Verification Problem 1.2. *Given a system defined by a system metamodel, an initial system state, specification of system behavior, and restrictions on the state space and given a set of safety properties, does every state in the restricted state space of the system satisfy the safety properties?*

While we can specify most abstract elements of this verification elements in the same fashion as Verification Problem 3.1 (p. 46), we have to formalize the restriction of a system's state space. We use a graph constraint to achieve this; this type of constraint is called *guaranteed constraint.*

Definition 4.1 (state space under constraint [4]). *Given a graph grammar* $GG = (GTS, G_0)$ *with* $GTS = (\mathcal{R}, TG)$ *and a graph constraint* $\mathcal{H}$*, we define the grammar's* state space under $\mathcal{H}$ *as* $\text{REACH}(GG, \mathcal{H}) = \{G \mid \exists n (G_0 \Rightarrow^n_{\mathcal{R}} G)$ *such that all traversed graphs satisfy* $\mathcal{H}\}$.

In essence, the state space under a guaranteed constraint consists of exactly those states that are reachable via paths of transformations where all graphs involved satisfy the guaranteed constraint (hence the name). While undesired or dangerous system states in safety-critical systems should be modeled and verified as safety properties there are three classes of cases where an a priori exclusion of states via a guaranteed constraint is justified:

Type graph constraints. Type graphs as defined in Definition 2.8 (p. 21) do not take cardinality constraints or other restrictions into account, even though those will play an important role in most systems. Particularly where physical systems are concerned, cardinality constraints can be used to exclude physical impossibilities: there is no sense in checking safety issues for physically impossible states.

External assumptions. If a system is to be developed under given assumptions, states violating both safety properties and the assumptions should not be considered possible violations of the specification. Hence, system verification should discard possible safety violations where the assumptions are not satisfied.

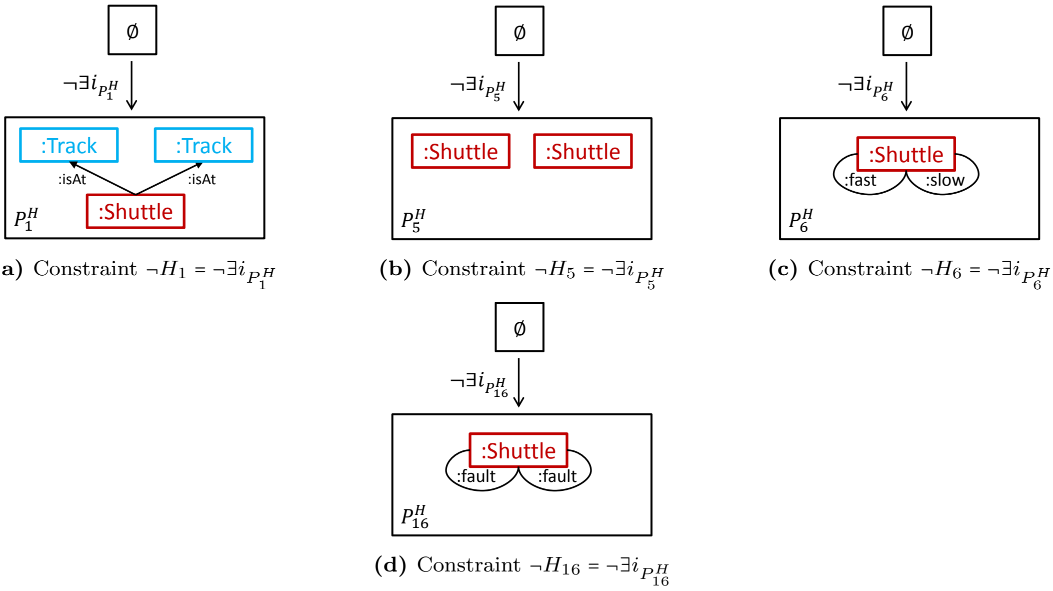
Controlled execution. Graph rules specify behavior and can include fine-grained conditions for their application; however, some systems may additionally be equipped with a higher-level controlling unit. For example, an interpreter used to execute graph transformation rules may include a preprocessing step that rejects the application of graph rules leading to states with certain properties.

Example 4.2 (guaranteed constraint). Figure 4.2(a) shows a typical example of a type graph constraint. More specifically, $\neg H_1$ is a cardinality constraint resulting from the physical impossibility of a shuttle being located on two tracks at the same time. We may want to have our systems' state spaces exclude states including such impossibilities.

Figure 4.2(b) is a special case of a type graph constraint and requires some explanation. The example system is supposed to specify a protocol for behavior of individual shuttles with respect to their speed modes – interaction of shuttles is not considered in this example. Hence, in order to avoid conflicts through concurrent behavior, the graph constraint $\neg H_5 = \neg\exists i_{P_5^H}$ guarantees that we anaylze behavior of shuttles only with respect to one individual shuttle at a time. The problem of concurrent behavior and its effect on inductive invariant checking is discussed in Chapter 9, Section 9.5.

Figure 4.2(c) is another example for a type graph restriction, although it is not a cardinality constraint. A shuttle can be in only one speed mode at a time; simultaneously driving fast and slow is not possible and is hence excluded by the graph constraint $\neg H_6 = \neg\exists i_{C_6}$.

The constraint $\neg H_{16}$ in Figure 4.2(d) is an example for an external assumption; specifically, a single fault assumption. In this example, we expect that it is possible to detect and count operational faults of shuttles. Those faults might include measurement errors, e.g. with respect to distance coordination, shuttle velocity, or failure to detect a switch ahead. If the probability of a shuttle encountering two such faults has been shown to fall beneath a certain threshold, the specification may include this single fault assumption as an external assumption governing



(a) Constraint $\neg H_1 = \neg\exists i_{P_1^H}$

(b) Constraint $\neg H_5 = \neg\exists i_{P_5^H}$

(c) Constraint $\neg H_6 = \neg\exists i_{P_6^H}$

(d) Constraint $\neg H_{16} = \neg\exists i_{P_{16}^H}$

Figure 4.2. – Fragments of a guaranteed constraint $\mathcal{H} = \neg H_1 \wedge \ldots \wedge \neg H_5 \wedge \neg H_6 \wedge \ldots \wedge \neg H_{16}$

the implementation of the system. Hence, the system will be implemented to ensure safety properties only under that assumption – and verification should likewise take the assumption into account.[1] In particular, assuming an extended type graph (by an additional edge type fault), we can model operational faults in shuttles by adding a fault edge per fault to the respective shuttle. The graph constraint $\neg H_{16}$ then forbids the existence of a shuttle with two fault edges – i.e., two failures – therey implementing the single fault assumption.

In our example system, the constraints $\neg H_1$, $\neg H_5$, $\neg H_6$, and other constraints not shown here are conjunctively combined as $\mathcal{H} = \neg H_1 \wedge \neg H_2 \wedge \neg H_3 \wedge \ldots \wedge \neg H_{15}$. While graph constraints can be arbitrarily complex with respect to nesting and boolean combinations, guaranteed constraints in our example systems are usually combinations of less complex existential conditions.

It is difficult to find an example for controlled execution in this example system, or in any physical system modeled as a graph transformation system. A preprocessing step requires computation of a transformation's result graph before its actual application, which is not feasible in terms of cost and performance in most physical or real-time systems. Transformation engines specifically tailored towards model transformations may take such preprocessing steps. A typical use case would be to ensure certain cardinality constraints or to remove temporary elements – markers, traceability information – after completing the model transformation. △

Even for cases outside of these classes, using a guaranteed constraint may make sense: sometimes, validity of properties in a system's state space has already been proven separately. Then, we can make use of this information during the verification of the remaining properties by using the established properties as a guaranteed constraint.

Note that the guaranteed constraint $\mathcal{H}$ has direct implications for the validity of $\mathcal{F}$ in state spaces. If, for example, $\mathcal{H}$ implies $\mathcal{F}$, $\mathcal{F}$ will obviously hold in all states of state spaces under $\mathcal{H}$ – independent of the behavior specified by the graph transformation system.

Given guaranteed constraints and their effect on state spaces as established by Definition 4.1 (p. 55), we will also extend the definition of operational invariants to reflect the effect of a guaranteed constraint:

[1] There are more than a few legal and ethical considerations involved in such an approach, especially where safety-critical systems are concerned, but their discussion is beyond the scope of this thesis.

Definition 4.3 (operational invariant)**.** *Given a graph grammar GG and graph constraints $\mathcal{F}$ and $\mathcal{H}$, we say that $\mathcal{F}$ is an* operational invariant of GG under $\mathcal{H}$*, if, for all graphs $G \in \mathrm{REACH}(GG, \mathcal{H})$, we have $G \models \mathcal{F}$.*

With that, we can refine Verification Problem 1.2 (p. 13) with appropriate formal elements:

Verification Problem 4.1. *Given a graph transformation system $GTS = (TG, \mathcal{R})$, a graph grammar $GG = (GTS, G_0)$, and graph constraints $\mathcal{F}$ and $\mathcal{H}$, is $\mathcal{F}$ an operational invariant of GG under $\mathcal{H}$?*

This verification problem is an extension of Verification Problem 3.1 (p. 46); conversely, the latter is a special case of the former: if we choose $\mathcal{H} = \mathrm{true}$, the two problems are identical.

While the use of guaranteed constraints to restrict state spaces of graph grammars makes sense in a number of situations, this approach raises certain challenges for system verification: it is desirable to have the graph transformation system or graph grammar under verification match the actual system as close as possible. Hence, the validity of guaranteed constraints should ideally be enforced by system behavior, i.e. the graph rules; then, it can be verified similar to the safety properties in question. This applies to both type graph constraints and constraints enforced by controlled execution, but not usually to external assumptions. In the approaches to follow, we will focus on the verification of safety properties under a guaranteed constraint. To also take explicit verification of the guaranteed constraint under consideration, we will introduce a verification approach for guaranteed properties at the end of this chapter. Its combination with the solutions to the verification problems above will then reason about the validity of safety properties without unverified assumptions about guaranteed constraints.

Furthermore, we need to modify our Definition of 1-inductive invariants to take the guaranteed constraint into account. Since a guaranteed constraint restricts the state space relevant for verification of the system, inductive invariants should similarly focus on transformations between graphs satisfying the guaranteed constraint.

Definition 4.4 (1-inductive invariant under constraint [1])**.** *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and graph constraints $\mathcal{F}$ and $\mathcal{H}$, $\mathcal{F}$ is an* inductive invariant *of GTS* under $\mathcal{H}$*, if, for all rules $b \in \mathcal{R}$, the following condition holds:*

$$\forall G, G' ((G \Rightarrow_b G' \wedge G \models \mathcal{H} \wedge G' \models \mathcal{H}) \Rightarrow (G \models \mathcal{F} \Rightarrow G' \models \mathcal{F})$$

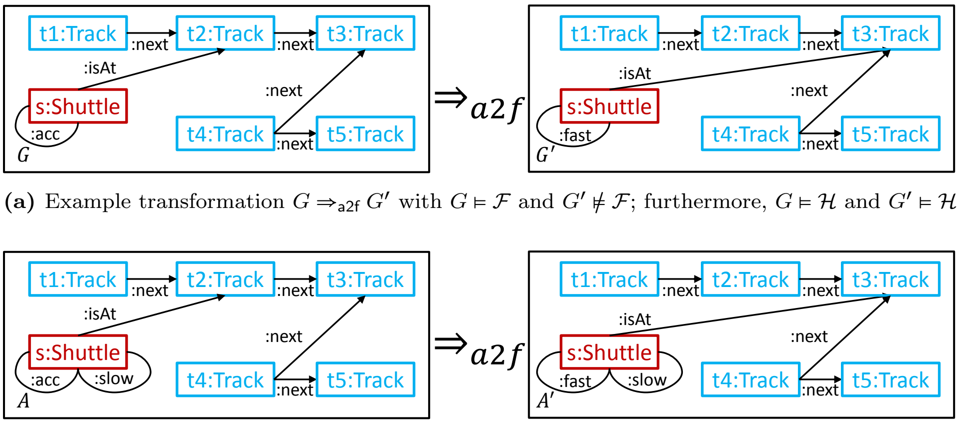
We also say that GTS preserves *$\mathcal{F}$* under *$\mathcal{H}$.*

This definition follows the idea explained earlier: violations of the guaranteed constraint can be excluded from our analysis. Here, $\mathcal{F}$ is only required to be preserved upon rule application if no violations of $\mathcal{H}$ are encountered before or after the transformation.

Similarly, Lemma 3.4 (p. 50), which reasons about the validity of $\mathcal{F}$ in the graph grammar's state space, has to be updated. We need to take the guaranteed constraint $\mathcal{H}$ into account: the approach now attempts to establish the validity of the safety property $\mathcal{F}$ as an operational invariant under $\mathcal{H}$ (i.e. in $\mathrm{REACH}(GG, \mathcal{H})$).

Lemma 4.5 (operational invariants of graph grammars under a constraint)**.** *Let $GG = (GTS, G_0)$ be a graph grammar with a graph transformation system $GTS = (TG, \mathcal{R})$ and let $\mathcal{F}$ and $\mathcal{H}$ be two graph constraints. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$, if the following conditions hold:*

1. *$G_0 \models \mathcal{F}$.*
2. *$\mathcal{F}$ is a 1-inductive invariant of GTS under $\mathcal{H}$.*



(a) Example transformation $G \Rightarrow_{\mathsf{a2f}} G'$ with $G \vDash \mathcal{F}$ and $G' \nvDash \mathcal{F}$; furthermore, $G \vDash \mathcal{H}$ and $G' \vDash \mathcal{H}$

(b) Example transformation $A \Rightarrow_{\mathsf{a2f}} A'$ with $G \vDash \mathcal{F}$ and $G' \nvDash \mathcal{F}$; however, $A' \nvDash \mathcal{H}$

Figure 4.3. – Example graph transformations as potential counterexamples for $\mathcal{F}$ being a 1-inductive invariant under $\mathcal{H}$

Proof. Consider a graph G in the grammar's state space under $\mathcal{H}$, i.e. $G \in \mathrm{REACH}(GG, \mathcal{H})$. Hence, there exists a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$ such that all traversed graphs satisfy $\mathcal{H}$. We will prove $G \vDash \mathcal{F}$ by induction over n:

Base case. For $n = 0$, we have $G = G_0$ and, by precondition, $G \vDash \mathcal{F}$.

Inductive step. Consider the sequence $G_0 \Rightarrow^{n+1}_{\mathcal{R}} G$ with $n + 1 > 0$. Then, there is a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$ such that all traversed graphs satisfy $\mathcal{H}$. By inductive hypothesis, we have $G' \vDash \mathcal{F}$. By Definition 4.4 and the existence of the transformation $G' \Rightarrow_{\mathcal{R}} G$ and since all traversed graphs satisfy $\mathcal{H}$ and $G' \vDash \mathcal{F}$, and since $\mathcal{F}$ is an 1-inductive invariant under $\mathcal{H}$, we have $G \vDash \mathcal{F}$, concluding the inductive proof. □

Example 4.6 (1-inductive invariant under constraint)**.** We recall the counterexample for $\mathcal{F}$ being a 1-indcutive invariant for GTS (Example 3.5, p. 50), which is shown again in Figure 4.3(a). Since both G and G' satisfy the guaranteed constraint $\mathcal{H}$, it is also a valid counterexample for $\mathcal{F}$ being a 1-inductive invariant under $\mathcal{H}$. On the other hand, consider the counterexample $A \Rightarrow_{\mathsf{a2f}} A'$ shown in Figure 4.3(b): since A' has a shuttle in speed modes slow and fast at the same time, it violates $\neg H_6$ (Figure 4.2(c)) and hence, violates the guaranteed constraint $\mathcal{H}$. Likewise, A violates a similar constraint. As a result, $A \Rightarrow_{\mathsf{a2f}} A'$ is not a valid counterexample when the guaranteed constraint is taken into account. $G \Rightarrow_{\mathsf{a2f}} G'$, however, still is.

With respect to Lemma 4.5, discarding counterexamples with violations of the guaranteed constraint $\mathcal{H}$ makes sense: we are interested in establishing the validity of the safety property in the state space $\mathrm{REACH}(GG, \mathcal{H})$, which does not include graphs violating $\mathcal{H}$. However, because of the counterexample $G \Rightarrow_{\mathsf{a2f}} G'$, $\mathcal{F}$ is not a 1-inductive invariant for GTS under $\mathcal{H}$ – we cannot use Lemma 4.5 to show validity of $\mathcal{F}$ in the graph grammar's state space. △

Lemma 4.5 solves Verification Problem 4.1 (p. 57). Like Verification Problem 3.1 (p. 46), it still addresses a singular graph grammar, but already considers the restricted state space unter a guaranteed constraint.

This brings us to the second issue highlighted at the end of Chapter 3: dealing with more than one initial state, i.e. start graph. We recall Verification Problem 1.3 (p. 13), which is an extension of Verification Problem 1.2 (p. 13):

Verification Problem 1.3. *Given a set of systems defined by a system metamodel, a set of initial states, specification of system behavior, and restrictions on the state space and given a* *set of safety properties, does every state in the restricted state spaces of all systems satisfy the safety properties?*

With guaranteed constraints, we have a formalization for restrictions imposed on state spaces. To formalize Verification Problem 1.3 (p. 13), we also need to supply a set of initial states for the systems in question. Considering all possible start graphs conforming to the type graph is neither feasible nor (usually) useful. For example, all start graphs violating a safety property will necessarily lead to an unsafe system (graph grammar). Thus, it makes sense to further restrict the set of possible start graphs. One the one hand, we would like to allow a large or possibly infinite number of start graphs; on the other hand, we need a symbolic representation of those start graphs such that their analysis ($G_0 \vDash \mathcal{F}$, the first condition of Lemma 3.4 (p. 50)) is still feasible. Our solution is the concept of induced graph grammars: a graph constraint, called a *start configuration constraint*, restricts the set of allowed start graphs. In combination with a graph transformation system, it induces a set of graph grammars.

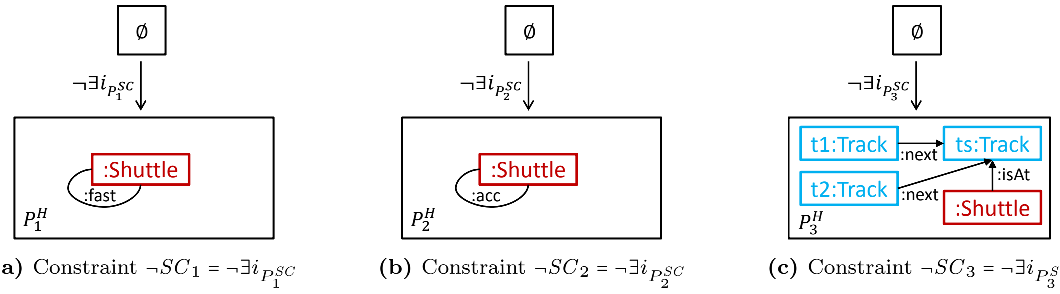
Definition 4.7 (induced graph grammars)**.** *Given a graph transformation system GTS and a graph constraint $\mathcal{S}$, we define the set of* graph grammars induced by *GTS and $\mathcal{S}$ as the set* $\text{IND}(GTS, \mathcal{S}) = \{(GTS, G_0) \mid G_0 \vDash \mathcal{S}\}$.

Here, $\mathcal{S}$ is not specified further, apart from the fact that it is a graph constraint. Note that similar to the relationship between a safety property $\mathcal{F}$ and a guaranteed constraint $\mathcal{H}$, the choice of the start configuration constraint $\mathcal{S}$ has implications for the validity of $\mathcal{F}$ and the state spaces.

In general, we will require the start configuration constraint $\mathcal{S}$ to imply the constraint $\mathcal{F}$ to be verified, i.e. $\mathcal{S} \vDash \mathcal{F}$. In other words, all possible start graphs need to be error-free. This can always be achieved: given a desired start configuration constraint $\mathcal{S}'$, we can choose the actual start configuration constraint as $\mathcal{S} = \mathcal{S}' \wedge \mathcal{F}$. This requirement will ease the proof obligations of our verification approach for the base of induction and shifts a part of that burden to system specification. For instance, if someone were to choose $\mathcal{S}'$ and $\mathcal{F}$ such that they contradict each other, $\mathcal{S} = \mathcal{S}' \wedge \mathcal{F}$ would not have any satisfying graphs and the set of induced graph grammars would be empty. However, finding such occurrences is not the focus of our verification approach: we want to verify that our systems are safe, not find out whether or not they fulfill the designer's intent outside of safety concerns. To give an extreme example, a designer also has the freedom to choose $\mathcal{F} = \text{true}$ or $\mathcal{H} = \text{false}$ – for the former case, all systems are safe by default, for the latter case, all state spaces are empty (which also makes all systems safe). Validating systems with respect to their specifications is not the focus here – we need to assume that constraints and parameters are suitably chosen. This, then, also applies to the choice of $\mathcal{S}$.

We do not usually have a similar requirement for $\mathcal{S}$ and $\mathcal{H}$. If $\mathcal{S}$ does not imply $\mathcal{H}$, there will be graph grammars in $\text{IND}(GTS, \mathcal{S})$ with empty state spaces under $\mathcal{H}$: in particular, for graph grammars GG with start graphs G_0 that satisfy $\mathcal{S}$ but violate $\mathcal{H}$, the state space of GG under $\mathcal{H}$ – i.e. $\{G \mid \exists n (G_0 \Rightarrow^n_{\mathcal{R}} G)$ such that all traversed graphs satisfy $\mathcal{H}\}$ – will evaluate to the empty set. Following the idea of a guaranteed constraint, these graph grammars are to be disregarded. This does not affect the verification result: by definition, these graph grammars will indeed have $\mathcal{F}$ as an operational invariant under $\mathcal{H}$, which is our verification goal. Hence, where verification of operational invariants under $\mathcal{H}$ is concerned, $\mathcal{S} \vDash \mathcal{H}$ is not required.

Example 4.8 (running example and start configuration constraint)**.** We recall Example 3.2 (p. 47) with Figures 3.2 and 3.3 (p. 49) where the shuttle example is modeled as a graph grammar with a type graph, a set of graph rules, and a start graph and where we have specified a safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2$.



(a) Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$ **(b)** Constraint $\neg SC_2 = \neg\exists i_{P_2^{SC}}$ **(c)** Constraint $\neg SC_3 = \neg\exists i_{P_3^{SC}}$

Figure 4.4. – Example start graph constraint $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$

For this example system, Figure 4.4 shows a graph constraint that could reasonably be chosen to describe start configurations. $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$ is a conjunction of three negated existential constraints. In particular, $\neg SC_1 = \neg\exists i_{P_1^{SC}}$ (Figure 4.4(a)) and $\neg SC_2 = \neg\exists i_{P_2^{SC}}$ (Figure 4.4(b)) specify that no shuttles in the system's initial state should be in speed modes fast or acc. In addition, $\neg SC_3 = \neg\exists i_{P_3^{SC}}$ (Figure 4.4(c)) forbids start graphs where a shuttle (regardless of its speed mode) is located on a switch. In summary, possible start graphs of the system are all graphs (typed over the type graph) that do not contain a shuttle on a switch and a shuttle driving in speed mode fast or acc.

Note that $\neg SC_3$ already implies the safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ from Example 3.2, Figure 3.3 (p. 49): if shuttles on a switch are forbidden (as part of the start graph), shuttles on a switch in mode fast, acc, or brake are necessarily forbidden as well. Furthermore, $\neg SC_1$ implies $\neg F_1$ and $\neg SC_2$ implies $\neg F_2$, for similar reasons.

Given the graph transformation system $GTS = (TG, \mathcal{R})$ and the graph G_0 specified in Example 3.2 (p. 47), we have $(GTS, G_0) \in \mathrm{IND}(GTS, \mathcal{S})$: G_0 indeed satisfies the start configuration constraint $\mathcal{S}$. △

With induced graph grammars and a start configuration constraint (and, as before, a guaranteed constraint), we can formalize Verification Problem 1.3 (p. 13) as follows:

Verification Problem VP.1g. *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and graph constraints $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{H}$ with $\mathcal{S} \models \mathcal{F}$, does every graph grammar $GG \in \mathrm{IND}(GTS, \mathcal{S})$ have $\mathcal{F}$ as an operational invariant under $\mathcal{H}$?*

In more detail, we ask whether the property ($\mathcal{F}$) holds in all states (G) of the state spaces under a guaranteed constraint ($\mathrm{REACH}(GG, \mathcal{H})$) for all graph grammars induced ($GG \in \mathrm{IND}(GTS, \mathcal{S})$) by the behavioral specification (GTS) and the restrictions on the start graphs ($\mathcal{S}$). The difference to Verification Problem 4.1 (p. 57) is the use of induced graph grammars and a start configuration constraint. We reason about $\mathrm{REACH}(GG, \mathcal{H})$ for graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$ as opposed to the state space (under $\mathcal{H}$) $\mathrm{REACH}(GG, \mathcal{H})$ of a singular graph grammar $GG = (GTS, G_0)$. The first formalized Verification Problem 3.1 (p. 46) in Chapter 3 considered neither induced graph grammars nor a guaranteed constraint.

As before, we introduce a solution to this verification problem:

Lemma 4.9 (operational invariants of induced graph grammars under a constraint)**.** *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and let $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \models \mathcal{F}$. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ for all graph grammars $GG \in$* $\mathrm{IND}(GTS, \mathcal{S})$*, if the following condition holds:*

2. $\mathcal{F}$ is a 1-inductive invariant of GTS under $\mathcal{H}$.

Proof. Consider an arbitrary graph grammar $GG \in \mathrm{IND}(GTS, \mathcal{S})$ and a graph G in the graph grammar's state space under $\mathcal{H}$, i.e. $G \in \mathrm{REACH}(GG, \mathcal{H})$. Hence, there exists a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$ such that all traversed graphs satisfy $\mathcal{H}$ and that $G_0 \vDash \mathcal{S}$. We will prove $G \vDash \mathcal{F}$ by induction over n:

Base case. For $n = 0$, we have $G = G_0$. By precondition, we have $\mathcal{S} \vDash \mathcal{F}$ and, with $G_0 \vDash \mathcal{S}$ and by Definition 2.36 (p. 42), $G_0 \vDash \mathcal{F}$.

Inductive step. Consider the sequence $G_0 \Rightarrow^{n+1}_{\mathcal{R}} G$ with $n + 1 > 0$. Then, there is a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$ such that all traversed graphs satisfy $\mathcal{H}$. By inductive hypothesis, we have $G' \vDash \mathcal{F}$. Since all traversed graphs satisfy $\mathcal{H}$ and by Definition 4.4, the existence of the transformation $G' \Rightarrow_{\mathcal{R}} G$, and $G' \vDash \mathcal{F}$ and since $\mathcal{F}$ is an 1-inductive invariant under $\mathcal{H}$, we have $G \vDash \mathcal{F}$, concluding the inductive proof. □

Lemma 4.9 then solves Verification Problem VP.1g by extending Lemma 4.5 (p. 57). Note that, similar to the definitions of inductive invariants with and without a guaranteed constraint, there is no need to change the proof's inductive step in comparison to Lemma 4.5 (p. 57) – it is unaffected by the notion of induced graph grammars and the start configuration constraint. Our argument for the inductive step is based on 1-inductive invariants, which refer only to the graph transformation rules, not to initial states of graph grammars. The base of induction follows from our requirement that start configuration constraints always imply the constraint that will be verified ($\mathcal{S} \vDash \mathcal{F}$). As a result, an explicit condition (1) for the base case of the inductive argument is missing in this lemma.

Example 4.10 (operational invariants of induced graph grammars under a constraint)**.** Since the inductive step has not changed in comparison to Lemma 4.5 (p. 57) and since we have not modified any graph rules in our graph transformation system, $\mathcal{F}$ will still not be a 1-inductive invariant under $\mathcal{H}$ – and Lemma 4.9 cannot be applied to the example. However, we can reiterate that the start configuration constraint $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$ indeed implies $\mathcal{F}$ as required – we have established this connection in Example 4.8 (p. 59). △

Verification Problem VP.1g and Lemma 4.9 have addressed the first and second issue brought up in Sections 1.2 and 1.3 and reiterated at the end of Chapter 3: taking additional information and restrictions of the state space into account and supporting multiple – possibly infinitely many – initial states. We are left with issues concerning the quality of the verification results: locality and lack of context information in our inductive argument. This is a side effect of using a symbolic approach; however, a symbolic approach is necessary to reason about infinitely many cases in finite time. In order to keep its positive and alleviate its negative effects, we will extend the notion of 1-inductive invariant to k-inductive invariants *[3]*.

The introduction of guaranteed constraints and induced graph grammars with start configuration constraints also leads to an extension of the original formal model (from Chapter 2) to what will be called the general formal model:

Formal Model (general)**.** *Systems and system specifications consist of the following elements:*

System metamodels *are specified by type graphs (Definition 2.8 (p. 21)).*

System states *– including initial states – are described by typed graphs (Definitions 2.1 (p. 20) and 2.8 (p. 21)).*

System behavior *is described by a typed graph transformation system (Definition 2.24 (p. 34)), which consists of typed graph transformation rules (Definition 2.20 (p. 32)).*

Properties *are modeled as graph constraints (Definition 2.12 (p. 25)) – this includes the constraint $\mathcal{F}$ to be verified, the guaranted constraint $\mathcal{H}$, and the start configuration constraint $\mathcal{S}$. As an additional requirement, $\mathcal{S}$ needs to imply $\mathcal{F}$, i.e. $\mathcal{S} \vDash \mathcal{F}$.*

Table 4.1. – Earlier and general formal model

Element	Formal Model (basic, earlier work)	Formal Model (general)
System metamodels	Type graphs	Type graphs
System states	Typed graphs	Typed graphs
System behavior	Graph rules with left application conditions	Graph rules with left application conditions
Safety properties	Graph constraint	Graph constraint
Guaranteed properties	–	Graph constraint
Set of initial states	No explicit concept; implicitly, same as the safety properties	Graph constraint
Systems	Typed graph grammars	Typed graph grammars
System state spaces	Graph grammars' state spaces	Graph grammars' state spaces a under guaranteed constraint
System sets	–	Induced graph grammars

Systems *are specified by typed graph grammars (Definition 2.28 (p. 36)), which consist of an initial state – a start graph – and a typed graph transformation system.*

System state spaces *are described by the state spaces (Definitions 2.28 (p. 36) and 4.1, p. 55) – the set of all reachable graphs – of the corresponding graph grammars (under the guaranteed constraint).*

System sets *are described by graph grammars induced (Definition 4.7 (p. 59)) by a graph transformation system and a start configuration constraint.*

The differences to the formal model used in earlier work (without considering restrictions in implementation) are highlighted in Table 4.1.

4.2. k-Induction

Intuitively, k-inductive invariant checking takes paths of transformations rather than singular transformations into account in order to accumulate more information about the situation and system in question and reduce the number of false negatives.

Definition D.1 (k-inductive invariant [3]). *Given a typed graph transformation system $GTS = (TG, \mathcal{R})$ and graph constraints $\mathcal{F}$ and $\mathcal{H}$, $\mathcal{F}$ is a k-*inductive invariant *of GTS* under *$\mathcal{H}$, if, for all sequences of transformations to $\mathcal{R}$ $trans = G_0 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_{k-1} \Rightarrow_{\mathcal{R}} G_k$ it holds that:*

$$(\forall z(0 \leq z \leq k \Rightarrow G_z \vDash \mathcal{H}) \land \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})) \quad \Rightarrow \quad (G_k \vDash \mathcal{F})$$

This definition formalizes the general notion of k-inductive invariants introduced in Definition 1.12 (p. 14). Intuitively, a graph constraint $\mathcal{F}$ is a k-inductive invariant if its validity in a path of transformations of length $k-1$ ($G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_{k-1}$) and $\forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})$ implies its validity after another singular transformation ($G_{k-1} \Rightarrow_{\mathcal{R}} G_k$ and $G_k \vDash \mathcal{F}$).

As before, only transformations where graphs satisfy the guaranteed constraint are considered ($\forall z(0 \le z \le k \Rightarrow G_z \vDash \mathcal{H})$).

Similar to the basic definition of 1-inductive invariants, the concept of k-inductive invariants can also be defined without a guaranteed constraint. We can, however, achieve the same effect using a guaranteed constraint $\mathcal{H}$ = true; hence, we will not establish a separate definition.

Note that 1-inductive invariants are a special case of k-inductive invariants *[3]*; conversely, k-inductive invariants are a generalization of 1-inductive invariants. For $k = 1$, the notion of k-inductive invariants is equivalent to the concept of 1-inductive invariants. Therefore, and to avoid confusion of the two terms, we use the term inductive invariant to refer to k-inductive invariants, including cases with $k = 1$; the earlier concept of inductive invariants *[1]* as in Definition 4.4 (p. 57) will be referred to as 1-inductive invariants.

Since we replace 1-inductive invariants with k-inductive invariants as the inductive step of an inductive proof, we also have to change the base case. For example, a 2-inductive invariant does not only require validity in a single state to conclude validity in subsequent steps, but requires validity in a path of length 1 (i.e., in two states). In more general terms, a k-indcutive invariant requires a base of induction of length $k - 1$. In order to express this condition in formal terms, we will first establish a state space that only encompasses states reachable from a graph grammar's initial state in a limited amount of steps, similar to bounded model checking [BCC⁺03].

Definition 4.11 (k-bounded state space under constraint *[4]*)**.** *Given a graph grammar* $GG = ((TG, \mathcal{R}), G_0)$ *and a number* $k \in \mathbb{N}$ *with* $k \ge 1$*, we define the graph grammar's* state space bounded by k*, or* k-bounded state space*, as* $\text{REACH}_k(GG) = \{G \mid \exists n(0 \le n \le k \wedge G_0 \Rightarrow^n_{\mathcal{R}} G)\}$*. We define the* state space bounded by k under *a graph constraint* $\mathcal{H}$ *as* $\text{REACH}_k(GG, \mathcal{H}) = \{G \mid \exists n(0 \le n \le k \ \wedge\ G_0 \Rightarrow^n_{\mathcal{R}} G)$ *such that all traversed graphs satisfy* $\mathcal{H}\}$*.*

Furthermore, we define $\text{REACH}_0(GG) = \{G_0\}$ *and* $\text{REACH}_0(GG, \mathcal{H}) = \{G_0\}$ *if* $G_0 \vDash \mathcal{H}$ *and* $\text{REACH}_0(GG, \mathcal{H}) = \varnothing$ *otherwise.*

Note that the replacement of 1-induction by k-induction is orthogonal to the extension from singular graph grammars to induced graph grammars. First, with the concept of bounded state spaces, we will extend the base case of Lemma 4.5 (p. 57), which reasons about a single graph grammar's state space (under a guaranteed constraint). Following that, we will combine both k-induction and induced graph grammars.

Lemma 4.12 (operational invariants of graph grammars under a constraint *[4]*)**.** *Let* $GG = (GTS, G_0)$ *be a graph grammar with a graph transformation system* $GTS = (TG, \mathcal{R})$ *and let* $\mathcal{F}$ *and* $\mathcal{H}$ *be two graph constraints.* $\mathcal{F}$ *is an operational invariant of* GG *under* $\mathcal{H}$*, if there exists a* $k \in \mathbb{N}$ *with* $k \ge 1$ *such that the following conditions hold:*

1. $\forall G(G \in \text{REACH}_{k-1}(GG, \mathcal{H}) \Rightarrow G \vDash \mathcal{F})$.
2. $\mathcal{F}$ *is a* k*-inductive invariant for* GTS *under* $\mathcal{H}$.

Proof. Consider a graph G in the graph grammar's state space under the constraint, i.e. $G \in \text{REACH}(GG, \mathcal{H})$. Hence, there exists a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$ such that all traversed graphs satisfy $\mathcal{H}$. If $n \le k - 1$, we have $G \vDash \mathcal{F}$ by precondition.

We will prove the case $n > k - 1$ by induction:

Base case. For $n = k$, there exist a transformation sequence $G_0 \Rightarrow^{k-1}_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$. Also, all graphs in the sequence satisfy $\mathcal{H}$ and there exist transformation sequences (whose length is smaller than k) from G_0 to all graphs in the sequence such that all traversed graphs satisfy $\mathcal{H}$. Hence, by precondition (1), $G' \vDash \mathcal{F}$ and all graphs in the sequence satisfy $\mathcal{F}$. Then, since $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ (2) and holds in all graphs of the sequence $G_0 \Rightarrow^{k-1}_{\mathcal{R}} G'$ and since we have $G_0 \Rightarrow^{k-1}_{\mathcal{R}} G'$ and $G' \Rightarrow_{\mathcal{R}} G$, we get $G \vDash \mathcal{F}$.

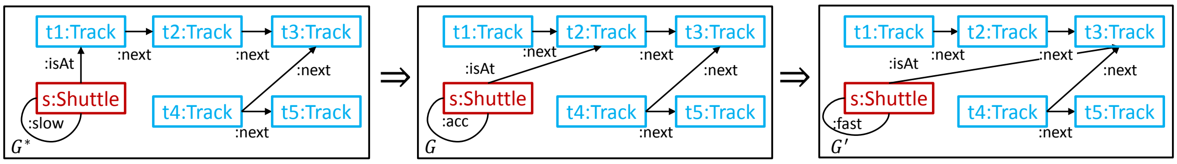


Figure 4.5. – Erroneous transformation sequence *trans* = $G^* \Rightarrow_{\mathsf{s2a}} G \Rightarrow_{\mathsf{a2f}} G'$

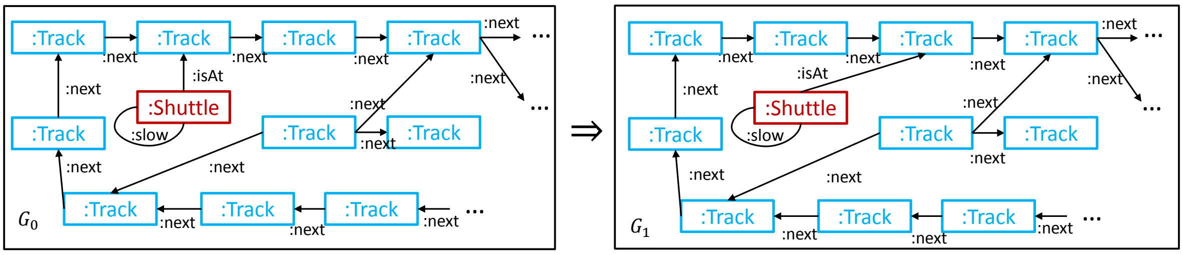


Figure 4.6. – Graph transformation $G_0 \Rightarrow_{\mathsf{s2s}} G_1$

Inductive step. Consider a sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$ with $n > k$. Then, there is a transformation sequence $G_0 \Rightarrow^{n-1}_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$. By inductive hypothesis, $\mathcal{F}$ holds in all graphs of the sequence $G_0 \Rightarrow^{n-1}_{\mathcal{R}} G'$. Since $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ (2) and $n > k$, we have $G \vDash \mathcal{F}$, concluding the inductive proof. □

Example 4.13 (2-inductive invariant and 1-bounded state space)**.** Given our example shuttle system (Example 3.2, Figures 3.2 and 3.3 (p. 49)), we already know that $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ is not a 1-inductive invariant under $\mathcal{H}$ – and, as a result, we cannot apply Lemmas 4.5 (p. 57) and 4.9 (p. 60). With the extension towards k-inductive invariants, however, we can attempt to verify $\mathcal{F}$ as a 2-inductive invariant – or, if that fails, as a k-inductive invariant for even higher values of k.

Consider the transformation $G \Rightarrow_{\mathsf{a2f}} G'$ used as a counterexample in Figure 4.3 (p. 58). In order to find out whether a similar counterexample exists for $\mathcal{F}$ as a 2-inductive invariant, we search for a transformation $G^* \Rightarrow_{\mathcal{R}} G$. Then, *trans* = $G^* \Rightarrow_{\mathcal{R}} G \Rightarrow_{\mathsf{a2f}} G'$ is a transformation sequence of length 2. By definition of k-inductive invariants, if we find that G^* satisfies both $\mathcal{H}$ and $\mathcal{F}$, *trans* is a counterexample since G' still violates $\mathcal{F}$.

Figure 4.5 shows such a sequence *trans* = $G^* \Rightarrow_{\mathsf{s2a}} G \Rightarrow_{\mathsf{a2f}} G'$. However, given a closer look, we notice that *trans* is not a valid transformation sequence: because of the negative application condition in s2a (Figure 3.2(h), p. 48), the rule is not applicable to G^*. Furthermore, there is no other possibility to reach G. Only the rule s2a can lead to a shuttle driving in speed mode acc; hence, $G \Rightarrow G'$ cannot be part of a proper counterexample.

Of course, this does not prove that $\mathcal{F}$ is a 2-inductive invariant for *GTS* under $\mathcal{H}$. It merely shows that we may have to consider other transformation sequences. Even without considering human error as a factor, it is usually difficult enough to find a counterexample by hand, assuming one exists. Manually proving the absence of counterexamples for infinitely many cases is impractible at least and usually impossible. This, then, is why we need an automated and symbolic algorithm.

Assuming that $\mathcal{F}$ is a 2-inductive invariant – which, as we will see later, it is – we still have to show the validity of $\mathcal{F}$ in the 1-bounded state space of *GG* under the guaranteed constraint *H*, i.e. in $\text{REACH}_1(GG, \mathcal{H})$. Given the start graph G_0 (Figure 3.2(i), p. 48), there is only one rule applicable – s2s. This results in the transformation $G_0 \Rightarrow_{\mathsf{s2s}} G_1$, which is depicted in Figure 4.6 (with some parts of G_0 and G_1 left out). Since both G_0 and G_1 satisfy $\mathcal{H}$, we have $\text{REACH}_1(GG, \mathcal{H}) = \{G_0, G_1\}$. Then, since both G_0 and G_1 satisfy $\mathcal{F}$, we have successfully established the induction base for Lemma 4.12. If $\mathcal{F}$ is a 2-inductive invariant for GTS under $\mathcal{H}$, we can conclude that $\mathcal{F}$ is satisfied in all graphs in the graph grammar's state space under $\mathcal{H}$. In other words, $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$. △

Lemma 4.5 (p. 57), which reasons about the validity of a graph constraint in a singular graph grammar's state space (under a guaranteed constraint) using 1-inductive invariants, can now be considered a special case of Lemma 4.12 (p. 63): if we only consider the case $k = 1$, both lemmas are equivalent. The set $\text{REACH}_{k-1}(GG, \mathcal{H}) = \text{REACH}_0(GG, \mathcal{H})$ (for $k = 1$) in condition 1 contains only the graph grammar's start graph G_0, which reduces the condition to $G_0 \vDash \mathcal{F}$ – unless $G_0 \nvDash \mathcal{H}$, in which case the graph grammar's state space under the constraint is empty anyway.

Similar to verification via 1-inductive invariant checking (Lemmas 4.5 (p. 57) and 4.9 (p. 60)), we can extend Lemma 4.12 (p. 63) in order to reason about the set of induced graph grammars rather than about individual graph grammars. This results in Lemma L.1, which describes the main proof obligations our approach and implementation described in Chapters 5 and 6 will fulfill.

Lemma L.1 (operational invariants of induced graph grammars under a constraint)**.** *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and let $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \vDash \mathcal{F}$. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ for all graph grammars GG $\in$ $\text{IND}(GTS, \mathcal{S})$, if there exists a $k \in \mathbb{N}$ with $k \geq 1$ such that the following conditions hold:*

1. $\forall GG(GG \in \text{IND}(GTS, \mathcal{S}) \Rightarrow \forall G(G \in \text{REACH}_{k-1}(GG, \mathcal{H}) \Rightarrow G \vDash \mathcal{F}))$.
2. *$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.*

Proof. Consider an arbitrary graph grammar $GG = (GTS, G_0)$ with $GG \in \text{IND}(GTS, \mathcal{S})$ and a graph G in the graph grammar's state space under $\mathcal{H}$, i.e. $G \in \text{REACH}(GG, \mathcal{H})$. Hence, there exists a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$ such that all traversed graphs satisfy $\mathcal{H}$ and that $G_0 \vDash \mathcal{S}$. If $n \leq k-1$, we have $G \vDash \mathcal{F}$ by precondition.

We will prove $G \vDash \mathcal{F}$ for the case $n > k-1$ by induction:

Base case. For $n = k$, there exist a transformation sequence $G_0 \Rightarrow^{k-1}_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$. Also, all graphs in the sequence satisfy $\mathcal{H}$ and there exist transformation sequences (whose length is smaller than k) from G_0 to all graphs in the sequence such that all traversed graphs satisfy $\mathcal{H}$. Hence, by precondition (1), $G' \vDash \mathcal{F}$ and all graphs in the sequence satisfy $\mathcal{F}$. Then, since $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ (2) and holds in all graphs of the sequence $G_0 \Rightarrow^{k-1}_{\mathcal{R}} G'$ and since we have $G_0 \Rightarrow^{k-1}_{\mathcal{R}} G'$ and $G' \Rightarrow_{\mathcal{R}} G$, we get $G \vDash \mathcal{F}$.

Inductive step. Consider a sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$ with $n > k$. Then, there is a transformation sequence $G_0 \Rightarrow^{n-1}_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$. By inductive hypothesis, $\mathcal{F}$ holds in all graphs of the sequence $G_0 \Rightarrow^{n-1}_{\mathcal{R}} G'$. Since $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ (2) and $n > k$, we have $G \vDash \mathcal{F}$, concluding the inductive proof. □

As with Lemma 4.5 (p. 57) and Lemma 4.12 (p. 63), where the former is a special case of the latter, Lemma 4.9 (p. 60) can now be considered a special case of Lemma L.1: considering only the case $k = 1$ will reduce the first condition to $\mathcal{S} \vDash \mathcal{F}$, making the lemmas equivalent.

While the inductive step has changed between 1-induction to k-induction, it is unaffected by the notion of induced graph grammars, as seen in Lemmas 4.12 (p. 63) and L.1. In all cases, it remains independent from state spaces and a start configuration constraint. However, the base case of our inductive argument has changed with both k-induction and induced graph grammars. Here (for $k \geq 2$), it needs to cover paths of transformation starting from the initial graph – the guarantee $\mathcal{S} \vDash \mathcal{F}$ is not enough. Even worse, the base case is no longer independent from the induced graph grammars' graph transformation system. Establishing the validity of $\mathcal{F}$ in the $k-1$-bounded state space from a specific start graph G_0, as required by Lemma 4.12

(p. 63), may be performed by bounded model checking – for small k, the computational effort may remain reasonable [4]. However, this is not possible for an infinite number of start graphs specified by the start configuration constraint. Here, similar to the verification of k-inductive invariants, we require a symbolic algorithm.

Both Lemma 4.12 (p. 63) and Lemma L.1 only consider operational invariants under the guaranteed constraint $\mathcal{H}$ – they reason about $\mathrm{REACH}(GG, \mathcal{H})$. As mentioned before, this approach is valid if a) the guaranteed constraint models external assumptions that are part of the system's specification or b) if additional measures are in place ensuring the validity of $\mathcal{H}$, such as postprocessing for the case of controlled execution. In the absence of such measures for the latter case, the validity of $\mathcal{H}$ has to be established separately. This can be included in our verification scheme:

Lemma L.2 (operational invariants of induced graph grammars [4]). *Let $GG = (GTS, G_0)$ be a graph grammar with a graph transformation system $GTS = (TG, \mathcal{R})$ and let $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \models \mathcal{F}$ and $\mathcal{S} \models \mathcal{H}$. $\mathcal{F}$ is an operational invariant of* $\mathrm{REACH}(GG)$ *for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$, if the following conditions hold:*

0. $\mathcal{H}$ is a 1-inductive invariant for GTS.
1/2. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$.

Proof. Consider an arbitrary graph grammar $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$. Since, by implication of graph constraints, $G_0 \models \mathcal{H}$ and since $\mathcal{H}$ is a 1-inductive invariant for GTS (1), we have $REACH(GG) = REACH(GG, \mathcal{H})$. Then, with (2), we have $G \models \mathcal{F}$ for all graphs $G \in \mathrm{REACH}(GG)$. □

The odd numbering comes from aligning this lemma to earlier lemmas: condition (1/2) can be established by Lemma L.1 (p. 65), which has two conditions (1) and (2).

Note that we require $\mathcal{S} \models \mathcal{H}$ here, similar to $\mathcal{S} \models \mathcal{F}$ earlier. In general, this can be achieved by choosing a desired start configuration constraint $\mathcal{S}'$ and defining the final start configuration constraint as $\mathcal{S} = \mathcal{S}' \wedge \mathcal{H}$. As before, we argue our focus is verification of system safety, not validation of sensible – e.g. non-contradictory – choices in system specifications. Given $\mathcal{S} \models \mathcal{H}$, all possible start graphs satisfy the guaranteed constraint. If the guaranteed constraint is preserved by rule applications (as a 1-inductive invariant, condition (0)), we then have $\mathrm{REACH}(GG, \mathcal{H}) = \mathrm{REACH}(GG)$ for all induced graph grammars GG.

The main ideas behind the application of this slightly different scheme are a) that there may be, depending on the case, a necessity to separately verify the guaranteed constraint and b) that the verification effort for the guaranteed constraint $\mathcal{H}$ is expected to be lower than that for $\mathcal{F}$. Recall, for instance, the constraint in Example 4.2, Figure 4.2(a), p. 56, which prevents a shuttle from being located on two tracks at the same time. If we want our modeled (graph transformation) system to resemble the actual shuttle protocol as close as reasonably possible, we will want to establish the validity of cardinality constraints. Hence, graph rules should preserve type graph and cardinality constraints, i.e. they should be 1-inductive invariants of the graph transformation system. Of course, they should also be fulfilled in possible start graphs.

This lemma decouples verification of $\mathcal{F}$ under $\mathcal{H}$ and verification of the guaranteed constraint $\mathcal{H}$ itself. In theory, we could extend this separation even beyond the notion of one safety property and one guaranteed constraint: a constraint could be split into multiple fragments that are verified iteratively (using k-induction), with each step having the conjunction of established fragments as a guaranteed constraint. While this extension is outside the scope of this thesis in terms of implementation and precise formal description, it can be performed by repeated manual execution of said iterations.

4.3. Conclusion

This concludes the extension of our inductive verification approach and the formalization of k-inductive invariant checking for graph transformation systems. Lemma L.1 (p. 65) (or Lemma L.2 (p. 66), depending on the scenario) describes the main proof obligations necessary to establish safety of a set of induced graph grammars by k-inductive invariant checking. Both lemmas solve Verification Problem VP.1g (p. 60), which is a formalization of the generic Verification Problem 1.3 (p. 13) for graphs and graphs transformation systems. As such, both lemmas address state space restrictions. Both lemmas also support verification of sets of induced graph grammars, i.e. graph grammars defined by graph transformation systems and sets of initial states – start graphs – described by a graph constraint. When only a singular graph grammar with one start graph is concerned, Lemma 4.5 (p. 57) can be applied, although this application scenario is not the main focus of this thesis.

Chapter 5 will explain how those proof obligations can be verified by a symbolic algorithm. Chapter 6 will then do the same for a restricted formal model, which allows for optimizations regarding computational effort and performance. To reiterate, our symbolic algorithms need to solve the following questions: given a graph transformation system GTS, a value $k \geq 1$, and graph constraints $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ with $\mathcal{S} \vDash \mathcal{F}$:

k−1-**bounded model checking:** for all graph grammars $GG \in \text{IND}(GTS, \mathcal{S})$ and all graphs $G \in \text{REACH}_{k-1}(GG, \mathcal{H})$, does $G \vDash \mathcal{F}$ hold?

k-**inductive invariant:** is $\mathcal{F}$ a k-inductive invariant for GTS under $\mathcal{H}$?

5. General Approach to k-Inductive Invariant Checking

Based on the formal foundations introduced in Chapter 2 and the verification approach of k-inductive invariant checking established in Chapter 4, we now define the required constructions and theorems to implement a corresponding verification algorithm and to prove its correctness.

We recall the general formal model established in Chapter 4:

Formal Model (general)**.** *Systems and system specifications consist of the following elements:*

System metamodels *are specified by type graphs (Definition 2.8 (p. 21)).*

System states *– including initial states – are described by typed graphs (Definitions 2.1 (p. 20) and 2.8 (p. 21)).*

System behavior *is described by a typed graph transformation system (Definition 2.24 (p. 34)), which consists of typed graph transformation rules (Definition 2.20 (p. 32)).*

Properties *are modeled as graph constraints (Definition 2.12 (p. 25)) – this includes the constraint $\mathcal{F}$ to be verified, the guaranted constraint $\mathcal{H}$, and the start configuration constraint $\mathcal{S}$. As an additional requirement, $\mathcal{S}$ needs to imply $\mathcal{F}$, i.e. $\mathcal{S} \vDash \mathcal{F}$.*

Systems *are specified by typed graph grammars (Definition 2.28 (p. 36)), which consist of an initial state – a start graph – and a typed graph transformation system.*

System state spaces *are described by the state spaces (Definitions 2.28 (p. 36) and 4.1 (p. 55)) – the set of all reachable graphs – of the corresponding graph grammars (under the guaranteed constraint).*

System sets *are described by graph grammars induced (Definition 4.7 (p. 59)) by a graph transformation system and a start configuration constraint.*

For the general approach discussed in this chapter, we do not impose additional restrictions on graph transformation systems or graph constraints. However, we will see that certain limitations to the formal model make sense in some scenarios and will discuss an approach for a restricted formal model in Chapter 6.

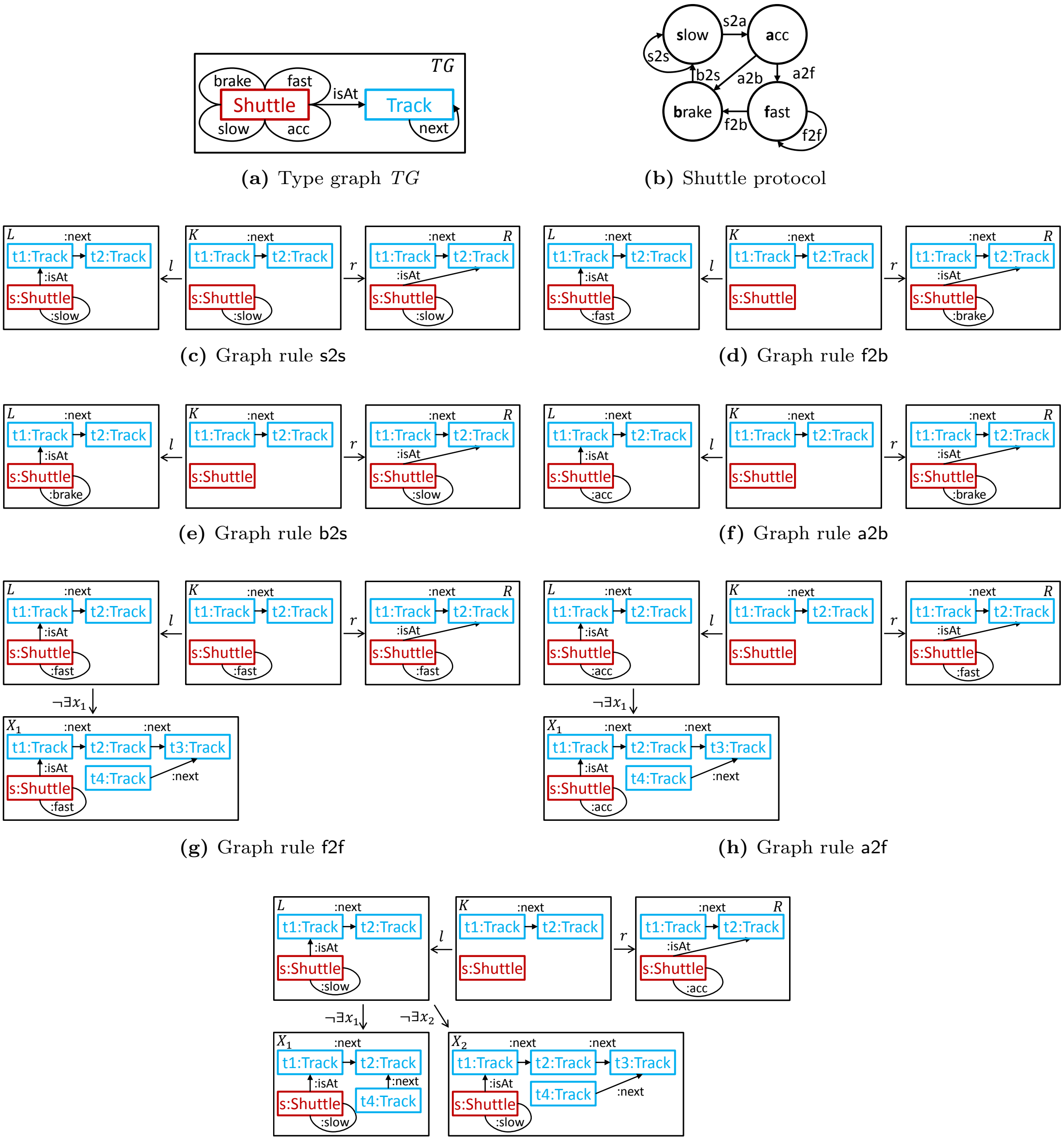
Given system specifications as listed above, we reiterate our central verification question from Chapter 4:

Verification Problem VP.1g. *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and graph constraints $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{H}$ with $\mathcal{S} \vDash \mathcal{F}$, does every graph grammar $GG \in \text{IND}(GTS, \mathcal{S})$ have $\mathcal{F}$ as an operational invariant under $\mathcal{H}$?*

Example 5.1 (running example)**.** For convenience, Figure 5.1 relists the type graph TG (Figure 5.1(a)), the speed mode transition protocol (Figure 5.1(b)), and the set of graph rules $\mathcal{R} = \{\mathsf{s2s}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{a2b}, \mathsf{f2f}, \mathsf{a2f}, \mathsf{s2a}\}$ (Figures 5.1(c)-5.1(i)) of our running example. All elements of the example are also listed in Sections C.1.2 and C.1.1 of Appendix C.

In Figure 5.2, we show three alternative rules for f2f, a2f, and s2a – f2f′, a2f′, and s2a′. These rules will lead to an unsafe system; we will use them occasionally in examples to demonstrate our verification approach. We will denote the associated graph transformation system and set of graph rules as $GTS' = (TG, \mathcal{R}')$ and $R' = \{\mathsf{s2s}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{a2b}, \mathsf{f2f'}, \mathsf{a2f'}, \mathsf{s2a'}\}$.

Figure 5.3 shows safety properties, the guaranteed constraint, and the start configuration constraint. We want to verify the permanent absence of shuttles driving on a switch in speed modes fast, acc, or brake (Figures 5.3(a)– 5.3(c)), i.e. our property to verify is $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$. Figures 5.3(d)-5.3(f) reiterate three fragments of the system's guaranteed constraint $\mathcal{H} =$



(a) Type graph *TG*

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

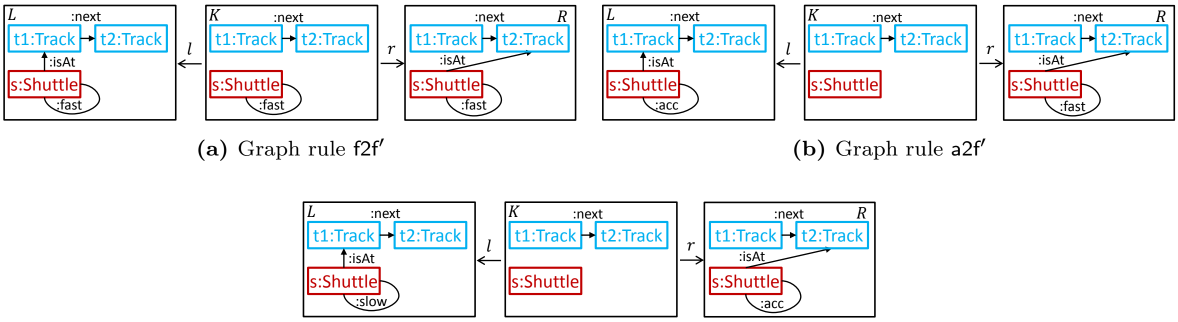
(f) Graph rule a2b

(g) Graph rule f2f

(h) Graph rule a2f

(i) Graph rule s2a

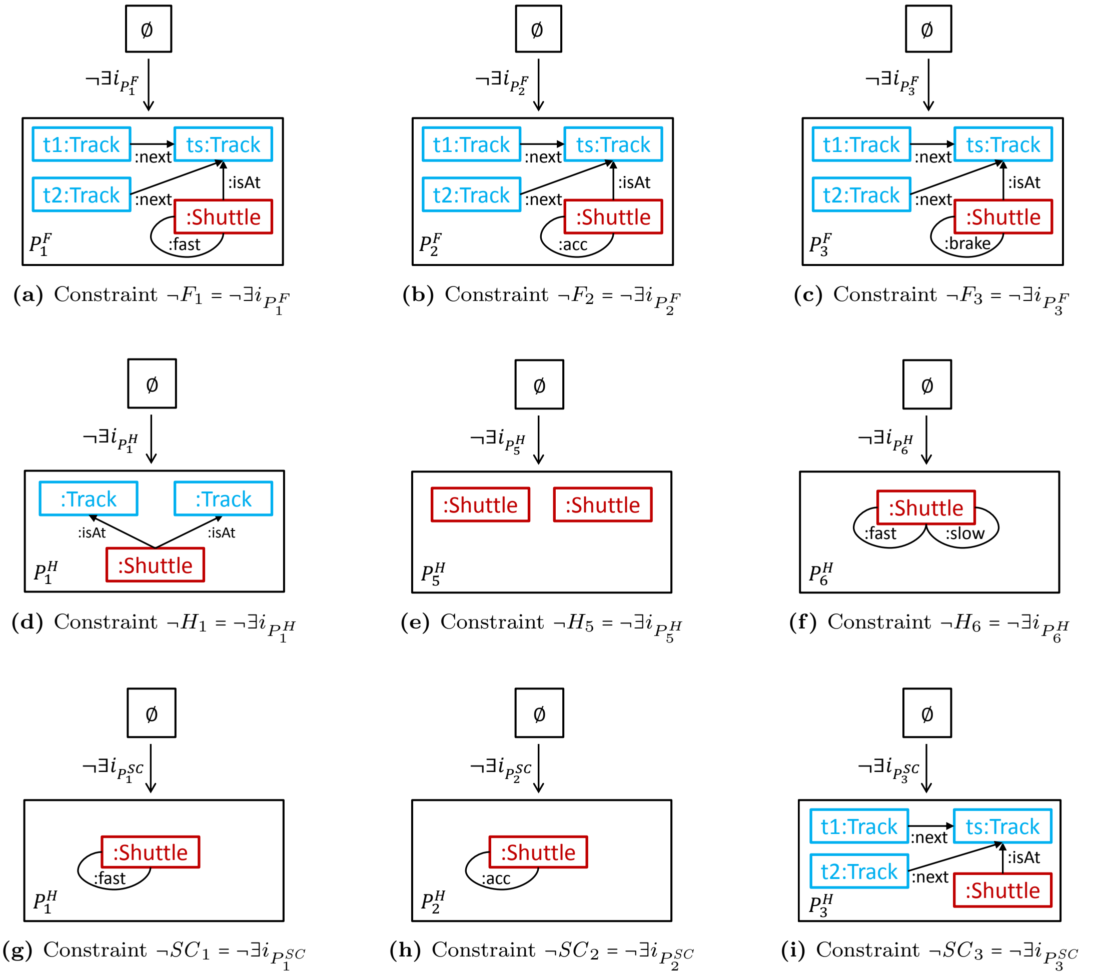
Figure 5.1. – Graph transformation system $GTS = (TG, \mathcal{R})$



(a) Graph rule f2f$'$ **(b)** Graph rule a2f$'$

(c) Graph rule s2a$'$

Figure 5.2. – Alternative rules f2f$'$, a2f$'$, and s2a$'$ without application conditions



(a) Constraint $\neg F_1 = \neg\exists i_{P_1^F}$ **(b)** Constraint $\neg F_2 = \neg\exists i_{P_2^F}$ **(c)** Constraint $\neg F_3 = \neg\exists i_{P_3^F}$

(d) Constraint $\neg H_1 = \neg\exists i_{P_1^H}$ **(e)** Constraint $\neg H_5 = \neg\exists i_{P_5^H}$ **(f)** Constraint $\neg H_6 = \neg\exists i_{P_6^H}$

(g) Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$ **(h)** Constraint $\neg SC_2 = \neg\exists i_{P_2^{SC}}$ **(i)** Constraint $\neg SC_3 = \neg\exists i_{P_3^{SC}}$

Figure 5.3. – Safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$, fragments of guaranteed constraint $\mathcal{H} = \neg\mathcal{H}_1 \wedge \ldots \wedge \neg H_{15}$, and parts of start configuration constraint $\mathcal{S}$

$\neg H_1 \wedge \ldots \wedge \neg H_{15}$. Finally, Figures 5.3(g), 5.3(h), and 5.3(i) show parts of the start configuration constraint $\mathcal{S}$. Depending on whether or not we limit verification to system states (and state spaces) under the guaranteed constraint $\mathcal{H}$, we can choose $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$ or $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3 \wedge \mathcal{H}$. In the latter case, we have $\mathcal{S} \vDash \mathcal{H}$ by construction. We can easily see that $\neg SC_1 \vDash \neg F_1$, $\neg SC_2 \vDash \neg \mathcal{F}_2$, and $\neg SC_3 \vDash \neg F_3$; then, $\mathcal{S} \vDash \mathcal{F}$ as required. If it were not obvious, we could join $\mathcal{F}$ conjuntively to $\mathcal{S}$ to form a new start configuration constraint.

With respect to the central verification question, we wonder whether all graphs in the state spaces of all graph grammars induced by the graph transformation system $GTS = (TG, \mathcal{R})$ (or $GTS' = (TG, \mathcal{R}'))$ and the start configuration constraint $\mathcal{S}$ fulfill the safety property. In other words, starting from a state satisfying our start configuration constraint and following the behavior specified by the graph rules, can the system reach a state where a shuttle reaches a switch while driving in mode fast, accelerating, or braking? △

We have already established that the size and potential infinity of the systems involved require a symbolic approach. Instead of exhaustively and explicitly analyzing systems' state spaces, we have chosen k-inductive invariant checking as our verification approach of choice. In Chapter 4, we have formalized k-inductive invariant checking for graph transformation systems and established proof obligations for its application in the form of Lemma L.1, reiterated here:

Lemma L.1 (validity of constraints in graph grammars). *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and let $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \vDash \mathcal{F}$. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$, if there exists a $k \in \mathbb{N}$ with $k \geq 1$ such that the following conditions hold:*

1. $\forall GG(GG \in \mathrm{IND}(GTS, \mathcal{S}) \Rightarrow \forall G(G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H}) \Rightarrow G \vDash \mathcal{F}))$.
2. *$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.*

Note that Verification Problem VP.1g (p. 60) and Lemma L.1 focus on operational invariants (and state spaces) under the restriction of a guaranteed constraint $\mathcal{H}$ without showing that constraint's validity. For the most part, the constructions and theorems for their formulaic implementation will follow the same idea. Similar to Chapter 4 and Lemma L.2 (p. 66), we will introduce an approach that combines results for restricted state spaces with explicit verification of $\mathcal{H}$ at the end of this chapter.

Outline. This chapter will introduce the formal notions and algorithmic constructions required to verify the lemma's two conditions in a finite fashion given the respective input elements GTS, $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$. As such, this chapter focuses on contribution **Formal-general** – the formalization of our general approach to verification with k-inductive invariants. Figure 5.4 provides a rough overview of this chapter's sections and the formal elements introduced.

In Section 5.1, we will first rearrange the condition for the inductive step of Lemma L.1 as the new Lemma 5.2. Likewise, Lemma 5.3 will rearrange the condition for the base of induction such that the condition is more intuitive from an algorithmic perspective. In particular, both lemmas will provide conditions that can be falsified by finding an appropriate transformation sequence as a counterexample.

However, falsifying or verifying the rearranged conditions will still require a symbolic encoding for a possibly infinite number of transformation sequences. Here, transformation sequences will be encoded in *k-sequences of source/target patterns* (Definition D.2), or *s/t-pattern sequences*. *S/t*-pattern sequences consist of a *source pattern*, a *target pattern*, and between 0 and $k-1$ *target/source patterns* (Definitions 5.7, 5.8, and 5.10) where k is the length of the sequence. All three types of patterns consist of application conditions over rule sides: thus, a source pattern encodes a match of a rule and its potential application, a target pattern encodes a comatch, and a target/source pattern encodes the combination of both. An *s/t*-pattern

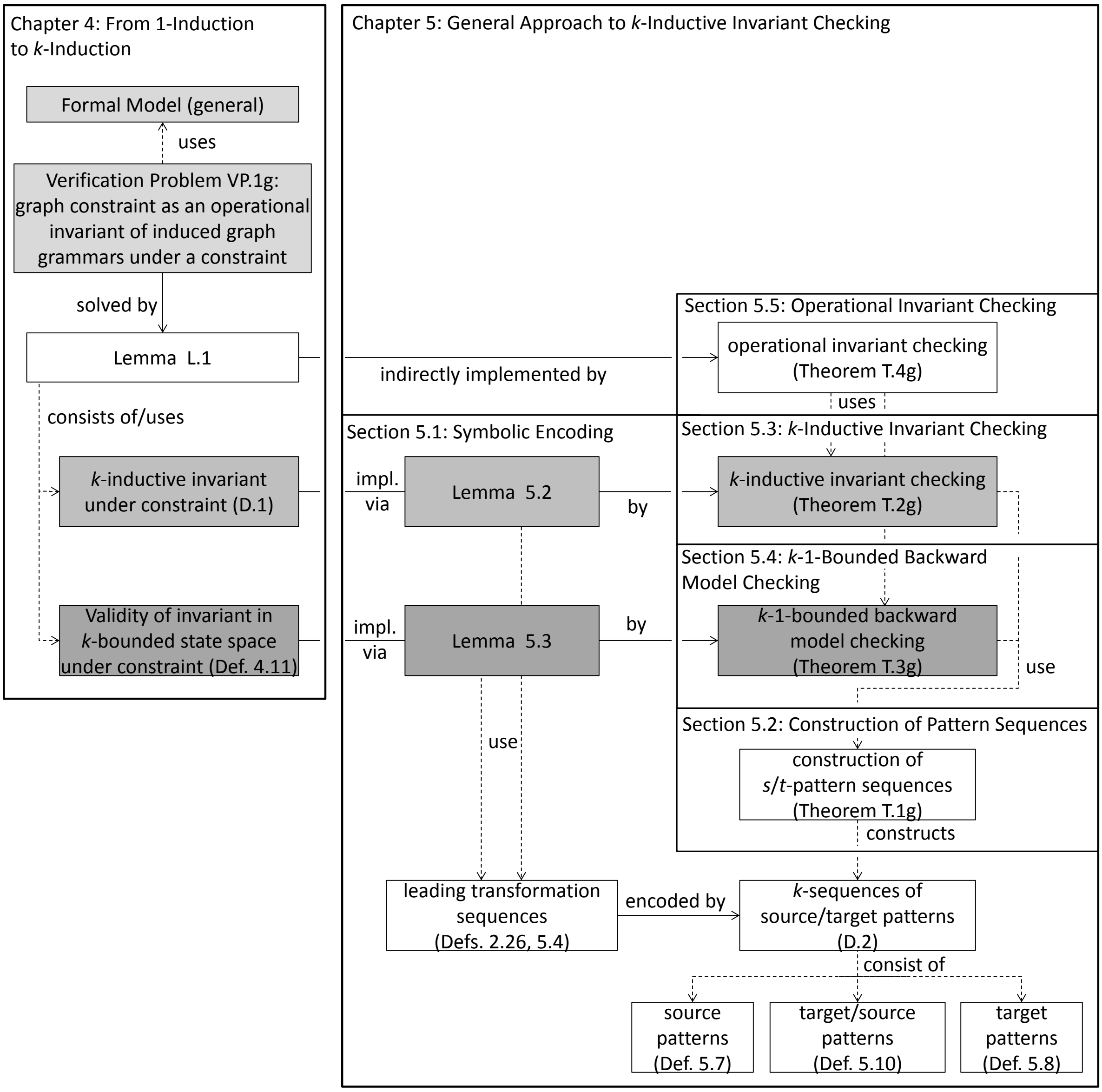

Chapter 4: From 1-Induction to *k*-Induction
Formal Model (general)
uses
Verification Problem VP.1g: graph constraint as an operational invariant of induced graph grammars under a constraint
solved by
Lemma L.1
consists of/uses
k-inductive invariant under constraint (D.1)
Validity of invariant in *k*-bounded state space under constraint (Def. 4.11)
Chapter 5: General Approach to *k*-Inductive Invariant Checking
indirectly implemented by
Section 5.5: Operational Invariant Checking
operational invariant checking (Theorem T.4g)
uses
Section 5.1: Symbolic Encoding
impl. via
Lemma 5.2
by
Section 5.3: *k*-Inductive Invariant Checking
k-inductive invariant checking (Theorem T.2g)
Section 5.4: *k*-1-Bounded Backward Model Checking
impl. via
Lemma 5.3
by
k-1-bounded backward model checking (Theorem T.3g)
use
use
Section 5.2: Construction of Pattern Sequences
construction of *s*/*t*-pattern sequences (Theorem T.1g)
constructs
leading transformation sequences (Defs. 2.26, 5.4)
encoded by
k-sequences of source/target patterns (D.2)
consist of
source patterns (Def. 5.7)
target/source patterns (Def. 5.10)
target patterns (Def. 5.8)

Figure 5.4. – Overview and dependencies of definitions, theorems, and lemmas

sequence then combines elements of the three types and can be satisfied by transformation sequences. Similar to a graph constraint encoding graphs or an application condition encoding its satisfying morphisms, an s/t-pattern sequence encodes transformation sequences of the same length that satisfy it.

Section 5.2 will explain how s/t-pattern sequences representing transformation sequences with certain properties can be created. Theorem T.1g introduces the Seq-construction, which takes a set of graph rules and two graph constraints as parameters. This construction is tailored to create s/t-pattern sequences (of a specified length) whose satisfying transformation sequences use rules from the given set of graph rules and fulfill certain properties: First, the transformation sequences *lead* to the first graph constraint passed to the construction, meaning that all rightmost graphs of the transformation sequences fulfill the graph constraint. Second, all intermediate and leftmost graphs fulfill the second constraint. With the right parameters, we can use the Seq-construction to create s/t-pattern sequences that (symbolically) represent all (and only those) transformation sequences that may serve as counterexamples for the conditions in Lemma 5.2 and Lemma 5.3.

However, s/t-pattern sequences, similar to application conditions and morphisms, may turn out to have no satisfying transformation sequences, i.e. to encode an empty set of transformation sequences. We will establish a lemma that provides a means of determining a s/t-pattern sequence's satisfiability: roughly speaking, if an interpretation of the leftmost source pattern as a graph constraint can be fulfilled by a graph, there exists a satisfying transformation sequence. That said, the satisfiability of a graph constraint (by any graph) is computationally challenging – and potentially undecidable [HP09]. Since the problem has been addressed in existing work [HP09, Pen09, SLO17, SLO18], its algorithmic perspective will not be discussed here.

In Section 5.3, Theorem T.2g then applies the Seq-construction (Theorem T.1g) and Lemma 5.14 to reason about the validity of a k-inductive invariant by verifying the condition established by Lemma 5.2. Likewise, Section 5.4 introduces Theorem T.3g, which reasons about the base case of our inductive invariant – the proposed invariant's validity in the induced graph grammars' k-bounded state space. Both theorems create a number of s/t-pattern sequences and then attempt to prove or disprove the existence of a satisfying graph for leftmost source patterns (by Lemma 5.14). The result expresses whether or not the constraint is a k-inductive invariant or is satisfied in the k-bounded state space of induced graph grammars, respectively.

Section 5.5 then combines k-inductive invariant checking and the invariant's validity in the k-bounded state space, i.e. the inductive step base case of our inductive verification approach. Theorem T.4g does not bring any new considerations, but uses the conditions and constructions of Theorem T.2g and Theorem T.3g. It establishes formal justification for our approach to verifying the conditions of Lemma L.1.

Finally, Section 5.6 discusses the results of this chapter and raises open issues.

5.1. Symbolic Encoding

As established, our verification idea follows an inductive approach: a graph constraint is established as a k-inductive invariant (inductive step) and checked for validity in the $k-1$-bounded state space from the graph grammars' start graphs (base case). Concerning the inductive step, we recall Definition D.1 for k-inductive invariants under a constraint:

Definition D.1 (k-inductive invariant *[3]*)**.** *Given a typed graph transformation system $GTS = (\mathcal{R}, TG)$ and graph constraints $\mathcal{F}$ and $\mathcal{H}$, $\mathcal{F}$ is a* k-inductive invariant *of GTS* under *$\mathcal{H}$, if, for all sequences of transformations to $\mathcal{R}$ trans $= G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ it holds that:*

$$(\forall z(0 \leq z \leq k \Rightarrow G_z \vDash \mathcal{H}) \land \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})) \quad \Rightarrow \quad (G_k \vDash \mathcal{F})$$

Thus, verification of a graph constraint as a k-inductive invariant relies on establishing the condition for all possible transformation sequences (Definition 2.26 (p. 36)) given all possible rules of the graph transformation system. In order to put the definition into a condition more directly verifiable by an algorithm, we will rearrange it in the form of the following lemma:

Lemma 5.2 (k-inductive invariant and transformation sequences as counterexamples)**.** *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and graph constraints $\mathcal{F}$ and $\mathcal{H}$, $\mathcal{F}$ is a k-*inductive invariant *of GTS* under *$\mathcal{H}$, if and only if there does not exist a transformation sequence to $\mathcal{R}$ trans $= G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ such that:*

$$G_k \nvDash \mathcal{F} \wedge G_k \vDash \mathcal{H} \wedge \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H})$$

Proof. We can rearrange the formula from Definition D.1 (p. 62) (for all sequences of transformations to $\mathcal{R}$ *trans* $= G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$):

$$\begin{aligned}
&(\forall z(0 \leq z \leq k \Rightarrow G_z \vDash \mathcal{H}) \wedge \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})) \Rightarrow (G_k \vDash \mathcal{F})\\
\Longleftrightarrow &\neg(\forall z(0 \leq z \leq k \Rightarrow G_z \vDash \mathcal{H}) \wedge \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})) \vee G_k \vDash \mathcal{F}\\
\Longleftrightarrow &\neg(G_k \vDash \mathcal{H} \wedge \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H})) \vee G_k \vDash \mathcal{F}\\
\Longleftrightarrow &G_k \vDash \mathcal{F} \vee \neg(G_k \vDash \mathcal{H} \wedge \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H}))\\
\Longleftrightarrow &\neg(G_k \nvDash \mathcal{F} \wedge G_k \vDash \mathcal{H} \wedge \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H}))
\end{aligned}$$

This last statement holding for all such transformation sequences is then equivalent to the absence of a sequence with $G_k \nvDash \mathcal{F} \wedge G_k \vDash \mathcal{H} \wedge \forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H})$. □

Similarly, we can rearrange the base case – i.e., the question about the validity of the invariant in the bounded state space from the graph grammars' start graphs – such that we gain a direct condition to be checked:

Lemma 5.3 (state spaces and transformation sequences as counterexamples)**.** *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \vDash \mathcal{F}$. For all graphs $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ with graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$, we have $G \vDash \mathcal{F}$ if and only if there does not exist a transformation sequence to $\mathcal{R}$ trans $= G_0 \Rightarrow_{b_1,m_1,m_1'} \dots \Rightarrow_{b_n,m_n,m_n'} G_n$ with $1 \leq n \leq k$ such that*

$$G_n \nvDash \mathcal{F} \wedge \forall i(1 \leq i \leq n \Rightarrow G_i \vDash \mathcal{H}) \wedge G_0 \vDash \mathcal{S}$$

Proof. Given a graph grammar $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$, by definition of k-bounded state spaces (Definition 4.11 (p. 63)), the existence of a graph $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ with $G \nvDash \mathcal{F}$ is equivalent to the existence of a transformation sequence to $\mathcal{R}$ *trans* $= G_0 \Rightarrow_{b_1,m_1,m_1'} \dots \Rightarrow_{b_n,m_n,m_n'} G_n$ with $1 \leq n \leq k-1$ and $G_n \nvDash \mathcal{F}$ where $G_0 \vDash \mathcal{S}$ and where all traversed graphs satisfy $\mathcal{H}$, i.e. $\forall i(1 \leq i \leq n \Rightarrow G_i \vDash \mathcal{H})$. The absence of such a sequence with $G_n \nvDash \mathcal{F} \wedge \forall i(1 \leq i \leq n \Rightarrow G_i \vDash \mathcal{H}) \wedge G_0 \vDash \mathcal{S}$ is then equivalent to validity of $\mathcal{F}$ in all $k-1$-bounded state spaces of the graph grammars induced by GTS and $\mathcal{S}$. □

Given Lemma 5.2, disproving a graph constraint as a k-inductive invariant requires a single transformation sequence fulfilling the respective condition as a counterexample. Likewise, Lemma 5.3 requires a counterexample to disprove validity of the constraint in question in the bounded state spaces. Conversely, the absence of such transformation sequences establishes the graph constraint $\mathcal{F}$ as a k-inductive invariant or as valid in the $k-1$-bounded state spaces, respectively. Note that if it were not for the requirement $\mathcal{S} \vDash \mathcal{F}$, we would have to add this condition in Lemma 5.3.

In both lemmas' conditions, the validity or violation of $\mathcal{F}$ in the transformation sequence's last (rightmost) graph has a prominent position. In order to capture that notion of validity in a formal manner, we define the notion of *leading* to a graph constraint.

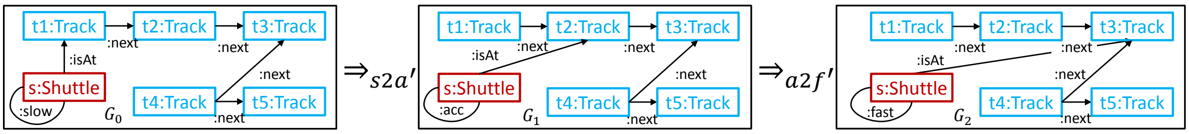


Figure 5.5. – Example transformation sequence *trans* leading to F_1 (and $\neg\mathcal{F}$) under $\mathcal{H}$

Definition 5.4 (leading [3])**.** *A transformation sequence trans* = $G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ leads *to a graph constraint* $\mathcal{C}$*, if* $G_k \vDash \mathcal{C}$*.*

Hence, both types of counterexamples are transformation sequences that lead to $\neg\mathcal{F}$. However, for transformation sequences to serve as counterexamples, we also require the validity of the guaraneed constraint $\mathcal{H}$ in all graphs of the sequence. Similar to the integration of guranteed constraints in state spaces and inductive invariants, we combine the notion of leading and guaranteed constraints in the following definition:

Definition 5.5 (leading under constraint)**.** *Given graph constraints* $\mathcal{C}$ *and* $\mathcal{H}$ *and a transformation sequence trans* = $G_0 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} G_k$*, we say that trans* leads to $\mathcal{C}$ *under* $\mathcal{H}$ *if* $G_k \vDash \mathcal{C}$ *and for all* i *with* $0 \leq i \leq k$*, we have* $G_i \vDash \mathcal{H}$*.*

Example 5.6 (leading, leading under constraint)**.** Figure 5.5 shows a transformation sequence *trans* = $G_0 \Rightarrow_{\mathsf{s2a}'} G_1 \Rightarrow_{\mathsf{a2f}'} G_2$. Given $\mathcal{F} = \neg F_1 \wedge \neg\mathcal{F}_2 \wedge \neg F_3$ (Example 5.1, Figures 5.3(a)–5.3(c), p. 71) and since we can find a fast shuttle on a switch as a subgraph of G_2, we have $G_2 \nvDash \neg F_1$, meaning that *trans* leads to F_1 and hence, to $\neg\mathcal{F}$.

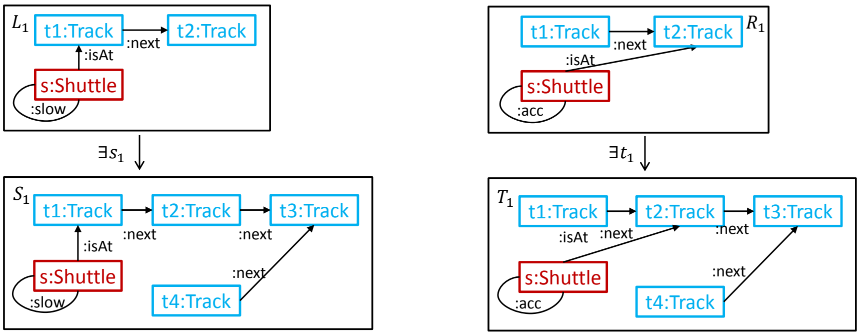
Furthermore, G_0, G_1, and G_2 satisfy $\mathcal{H}$ (Example 5.1, Figures 5.3(d)–5.3(f), p. 71). In particular, we cannot find the situations forbidden by $\mathcal{H}$ in G_0, G_1, or G_2. Thus, *trans* leads to $\neg F$ under $\mathcal{H}$.

As such, *trans* is already a counterexample for $\mathcal{F}$ being a 2-inductive invariant for $GTS' = (TG, \mathcal{R}')$ under $\mathcal{H}$. This would not work with $GTS = (TG, \mathcal{R})$ – the application conditions in s2a and a2f would not allow a transformation sequence $G_0 \Rightarrow_{\mathsf{s2a}} G_1 \Rightarrow_{\mathsf{s2a}} G_2$. Furthermore, *trans* is also a counterexample for the validity of $\mathcal{F}$ in all 2-bounded state spaces of induced graph grammars IND$(GTS', \mathcal{S})$: since $G_0 \vDash \mathcal{S}$, it is a valid start graph, and F_1 can be reached after two rule applications. △

With the definition of leading under a constraint, we can say that verification of k-inductive invariants in the sense of Lemma 5.2 amounts to finding transformation sequences to $\mathcal{R}$ leading to $\neg\mathcal{F}$ under H.

In order to verify this condition in a symbolic fashion, we require a symbolic encoding for transformation sequences. In principle, each graph rule is already a symbolic encoding for a number of graph transformations applying that rule. By extension, an ordered set of transformation rules symbolically encodes all possible applications of graph rules in the respective order. However, due to its genericity, such an encoding would encode only a minimum of information; our analysis requires more context to yield a sufficiently precise result: First, we do not want to consider all transformation sequences, but only those leading to a violation of $\mathcal{F}$, i.e. leading to $\neg\mathcal{F}$. Second, since a transformation sequence serving as a counterexample requires the validity of $\mathcal{H}$ in all traversed graphs, we are also interested in all intermediate graphs occurring in the sequence. Third, interactions between subsequent rule applications – in the form of possible overlappings between left and right rule sides – have a significant impact on the resulting graphs in the transformation sequence and need to be considered.

Therefore, we use source and target patterns as a means to encode context beyond left and right sides of a rule in the form of nested application conditions. In particular, a source pattern



(a) Source pattern $src = \exists s_1 : L_1 \hookrightarrow S_1$

(b) Target pattern $tar = \exists t_1 : R_1 \hookrightarrow T_1$

Figure 5.6. – Example source pattern $src = \exists s_1$ and target pattern $tar = \exists t_1$ over rule sides of s2a

describes context in which the left side of a rule – and hence, a match for the rule's application – can occur.

Definition 5.7 (source pattern)**.** *Given a graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$, a* source pattern *over b is an application condition over the left side L.*

Formally, a source pattern *src* over a rule b serves as a symbolic representative for all morphisms and potential matches $m : L \hookrightarrow G$ where the morphism satisfies the source pattern, i.e. where $m \vDash src$. This is a direct application of the notion of application conditions encoding (satisfying) morphisms; here, we specifically target rule matches and rule applications. Given a source pattern over b, we will sometimes refer to it as a source pattern over L instead – in particular, if the left rule side is of particular importance.

Similar to source patterns, target patterns encode situations in which the right side of a rule – a comatch for the rule's application – occurs. A target pattern *tar* (over a rule b or a right rule side R) serves as a representative for all morphisms and potential comatches $m' : R \hookrightarrow T$ where m' satisfies *tar*.

Definition 5.8 (target pattern)**.** *Given a graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$, a* target pattern *over b is an application condition over the right side R.*

Together, a source and target pattern over the left and right side of the same rule can encode a potential application of the rule – including additional context specified by the source and target pattern, i.e. the application conditions over L and R. Note that we did not place any restrictions on source and target patterns. Thus, it is possible to create unsatisfiable source or target patterns; for instance, $src = \text{false}$ would be a legitimate source pattern.

Example 5.9 (source and target patterns)**.** Consider the source pattern $src = \exists s_1$ and target pattern $tar = \exists t_1$ shown in Figures 5.6(a) and 5.6(b), respectively, with L_1 and R_1 being the left and right rule sides of the graph rule s2a′ (and also s2a). *src* embeds the left side L_1 into a context more specific than the left side itself; likewise, *tar* provides additional context beyond the rule's right side. In particular, the additional context consists of two additional tracks, one of which is a switch ahead of the shuttle.

The upper and lower thirds of Figure 5.7 show a transformation $G_0 \Rightarrow_{\mathsf{s2a}', m_1, m_1'} G_1$. Furthermore, the source and target pattern $src = \exists s_1$ and $tar = \exists t_1$ are depicted again in the Figure's upper and middle parts. Since there exists an injective morphism $y_1 : S_1 \hookrightarrow G_0$ with $y_1 \vDash s_1 = m_1$, the match m_1 satisfies *src*. Likewise, the existence of $y_1' : T_1 \hookrightarrow G_0$ such that $y_1' \circ t_1 = m_1'$ implies $m_1' \vDash tar$.

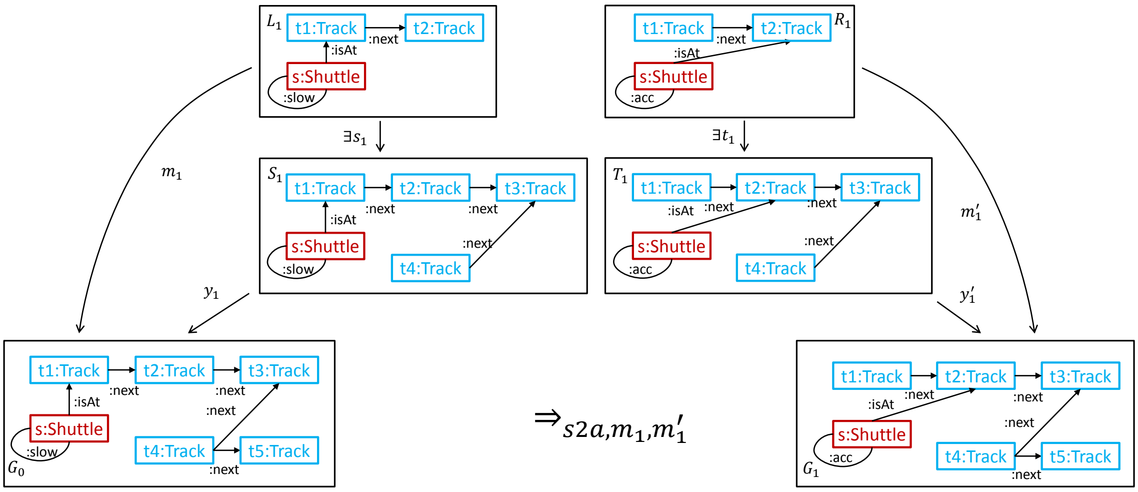


Figure 5.7. – Example transformation $G_0 \Rightarrow_{\mathsf{s2a}',m_1,m_1'} G_1$ matching source and target pattern

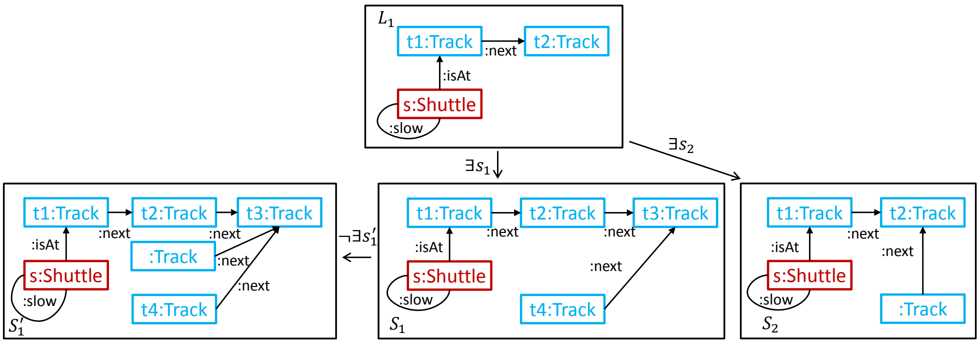


Figure 5.8. – Example source pattern $src' = \exists(s_1, \neg\exists s_1') \vee \exists s_2$

In particular, *src* and *tar* together describe potential applications of $\mathsf{s2a}'$ in the context of a switch two tracks ahead of the respective shuttle's current position. $G_0 \Rightarrow_{\mathsf{s2a}',m_1,m_1'} G_1$, as depicted, is one of infinitely many concrete rule applications encoded by *src* and *tar*.

Since source and target patterns are defined as application conditions, they can be as complex as nested application conditions. Consider, for example, the source pattern $src' = \exists(s_1, \neg\exists s_1') \vee \exists s_2$ depicted in Figure 5.8. It describes the occurrence of the left rule side in the context of a switch (t3) two tracks ahead ($\exists(s_1, \ldots)$) without a third track leading to the switch ($\neg\exists s_1'$) – or the context of the shuttle's subsequent track (t2) being a switch ($\ldots \vee \exists s_2$). △

The concept of source and target patterns has been applied in a more restricted fashion in previous work *[1]* concerned with the verification of 1-inductive invariants. In order to apply the idea and the symbolic encoding it entails to k-inductive invariant checking and bounded state spaces, we require not only singular source and target patterns (for singular transformations) but a sequence of source and target patterns (for a sequence of transformations) – compare the notions of 1-inductive invariants (Definition 4.4 (p. 57)) and k-inductive invariants (Definition D.1 (p. 62)).

However, it is not sufficient to combine sequences of pairs of source and target patterns. By doing so, we would disregard the fact that a comatch of one rule application can overlap the match of a subsequent rule application – or that their contexts (in the source and target pattern) can. Those combinations of target and source patterns are represented by *target/source patterns*.

Definition 5.10 (target/source pattern)**.** *Given rules $b_1 = \langle(L_1 \hookleftarrow K_1 \hookrightarrow R_1), ac_{L_1}, ac_{R_1}\rangle$ and $b_2 = \langle(L_2 \hookleftarrow K_2 \hookrightarrow R_2), ac_{L_2}, ac_{R_2}\rangle$ and a graph E with a pair of injective and jointly surjective morphisms $(e_R : R_1 \hookrightarrow E, e_L : L_2 \hookrightarrow E)$, a* target/source pattern *over (b_1, b_2) is a pair of application conditions over R_1 and L_2 of the form $(\exists(e_R, ac_E), \exists(e_L, ac_E))$ with ac_E being an application condition over E.*

A pair of morphisms with the same codomain $(m_1' : R_1 \hookrightarrow G, m_2 : L_2 \hookrightarrow G)$ satisfies a target/source pattern (tar, src), denoted $(m_1', m_2) \vDash (tar, src)$, if m_1' and m_2 satisfy tar and src via a common injective morphism, i.e. if there exists $y : E \hookrightarrow G$ with $y \vDash ac_E$, $y \circ e_R = m_1'$, and $y \circ e_L = m_2$.

$$\begin{tikzcd}
R_1 \arrow[dr, hook, "e_R"] \arrow[ddr, hook, bend right, "m_1'{=}y\circ e_R"'] & & L_2 \arrow[dl, hook', "e_L"'] \arrow[ddl, hook', bend left, "m_2{=}y\circ e_L"] \\
& E \triangleleft ac_E \arrow[d, hook, "y \vDash"] & \\
& G &
\end{tikzcd}$$

The specific overlapping of the right and left rule sides and their contexts in a target/source pattern is encoded in the graph E and the morphisms e_R and e_L. While a simple pairing of a target and a source pattern would represent all such overlappings, specifically representing one at a time in a target/source pattern allows for a more precise analysis, which is one of the main goals of extending verification with 1-inductive invariants to k-inductive invariants.

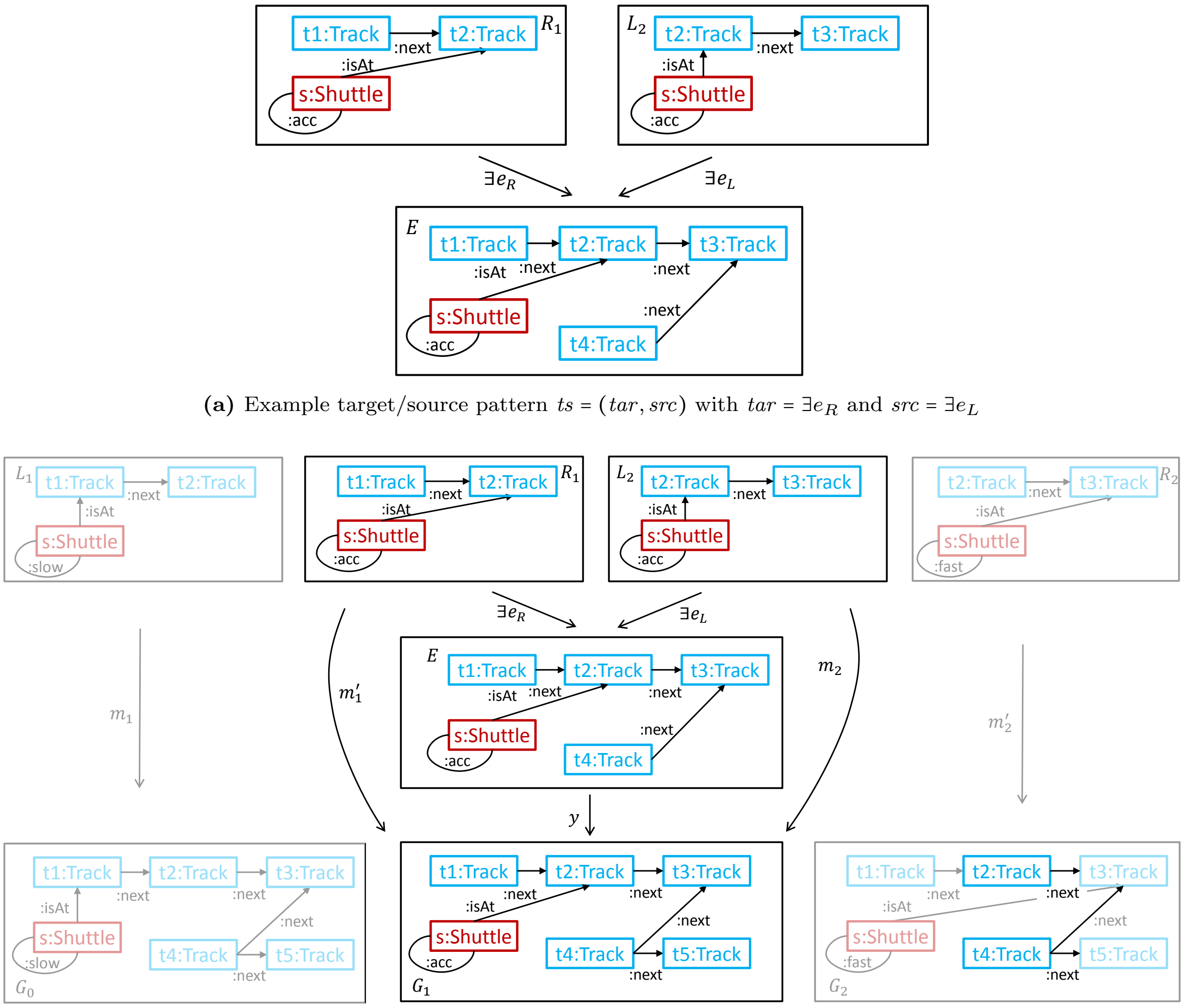
Example 5.11 (target/source patterns)**.** Figure 5.9(a) shows a target/source pattern $ts = (\exists(e_R : R_1 \hookrightarrow E), \exists(e_L : L_2 \hookrightarrow E))$ that describes context beyond the interaction of a potential comatch of a right rule side R_1 and a potential match of a left rule side L_2, which belong to graph rules $\mathsf{s2a}'$ and $\mathsf{a2f}'$, respectively.

Furthermore, Figure 5.9(b) shows a comatch/match pair (m_1', m_2) where there exists an injective morphism $y : E \hookrightarrow G_1$ such that $y \circ e_R = m_1'$ and $y \circ e_L = m_2$ and hence, (m_1', m_2) satisfies the target/source pattern *ts*. Note that G_1, m_1', and m_2 could be part of a transformation sequence $G_0 \Rightarrow_{\mathsf{s2a}', m_1, m_1'} G_1 \Rightarrow_{\mathsf{a2f}', m_2, m_2'} G_2$ when including the transparent parts of Figure 5.9(b).

With respect to the example, the target/source pattern *ts* symbolically describes a number of situations where the right side of rule $\mathsf{s2a}$ and the context it appears in relates to the left side of rule $\mathsf{a2f}$ and its context in a certain way: specifically, the potential subsequent application of both rules relate to the same shuttle, which – after the first rule application and before the second – is positioned one track ahead of a switch.

While symbolically representing a comatch/match pair and graph that is not significantly larger or more complex than the target/source pattern might not seem particularly impressive, note that the pair (m_1', m_2), the graph G_1, and the transformation sequence are just one example. There is – in this case – an infinite number of comatch/match pairs (and transformation sequences) such that the pair satisfies the target/source pattern *ts* – including significantly larger graphs not depicted here.

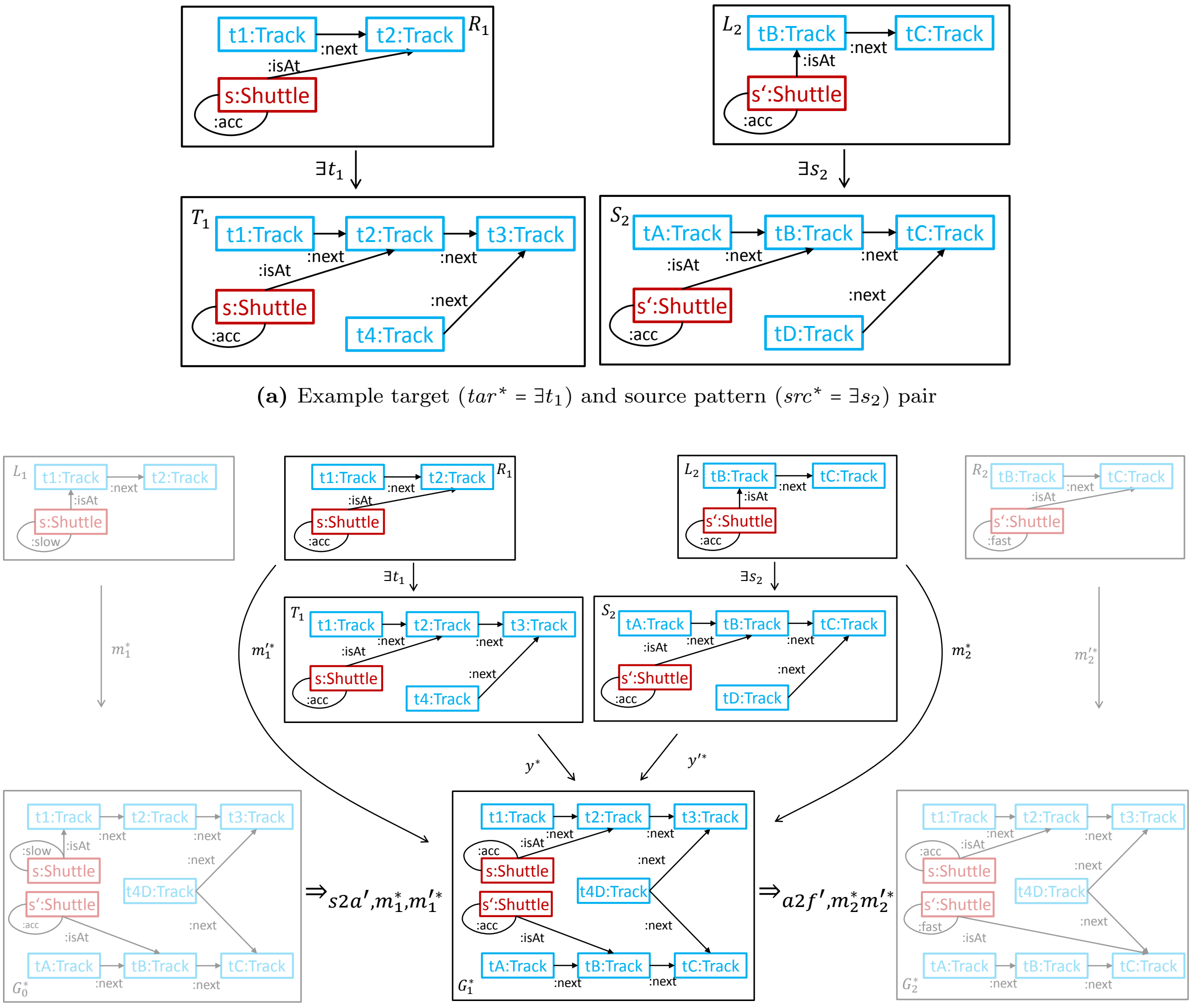
In order to better understand the difference between a target/source pattern and a pair of a target and a source pattern, consider Figures 5.10(a) and 5.10(b). The former depicts a pair of a target pattern $tar^* = \exists t_1$ (over the right side of rule $\mathsf{s2a}$) and a source pattern $src^* = \exists s_2$ (over the left side of rule $\mathsf{a2f}$) without specifying potential overlappings described by a target/source pattern. The latter figure shows a comatch/match pair $(m_1'^*, m_2^*)$ such that the comatch $m_1'^*$ satisfies tar^* (via y^*) and the match m_2^* satisfies src^* (via y'^*). Likewise,



(a) Example target/source pattern $ts = (tar, src)$ with $tar = \exists e_R$ and $src = \exists e_L$

(b) Example comatch/match pair (m_1', m_2) with $(m_1', m_2) \models ts$

Figure 5.9. – Example target/source pattern and satisfying comatch/match pair



(a) Example target ($tar^* = \exists t_1$) and source pattern ($src^* = \exists s_2$) pair

(b) Example comatch/match pair $(m_1'^*, m_2^*)$ with $m_1'^* \models tar^*$ and $m_2^* \models src^*$

Figure 5.10. – Example pair of target and source pattern and satisfying comatch/match pair

given the comatch/match pair (m_1', m_2) and graph G_1 from Figure 5.9(b), we could also find injective morphisms $x : T_1 \hookrightarrow G_1$ and $x' : S_2 \hookrightarrow G_1$ such that $x \circ t_1 = m_1'$ and $x' \circ s_2 = m_2$ and hence, $m_1' \vDash \mathit{tar}^*$ and $m_1 \vDash \mathit{src}^*$. However, the reverse is not true: the morphism pair $(m_1'^*, m_2^*)$ clearly does not satisfy the target/source pattern *ts* (see Figure 5.9(a)) because $m_1'^*$ and m_2^* relate to different shuttles while *ts* requires the shuttles in the right and left rule sides to refer to the same shuttle. △

The difference between a target/source pattern and a pair of a target and a source pattern makes sense because a pair of a target and a source pattern does not place any restrictions on potential overlappings of the rule sides or their context – conversely, a target/source pattern explicitly specifies such an overlapping by the morphism pair's $((e_R, e_L))$ common codomain E. As such, a target/source pattern is more restrictive. Given a pair of a target and a source pattern $(\mathit{tar}^*, \mathit{src}^*)$ and a set of morphism pairs $(m_1'^*, m_2^*)$ such that $m_1'^* \vDash \mathit{tar}^*$ and $m_2^* \vDash \mathit{src}^*$, considering all overlappings of tar^* and src^* would create a number of target/source patterns such that each of the aforementioned morphism pairs $(m_1'^*, m_2^*)$ would satisfy one of those target/source patterns.

While the above figures and examples already showed a target/source pattern and comatch/-match pairs appearing in the context of a transformation sequence, we still have not established an encoding for actual transformation sequences. In order to do so, we combine source patterns, target patterns, and target/source patterns to create *k-sequences of source/target patterns* – or *s/t-pattern sequences* for short.

Definition D.2 (k-sequence of source/target patterns [3])**.** *Given a* $k \geq 1$*, a source pattern* src_1 *over a rule* b_1*, a target pattern* tar_k *over a rule* b_k *and a number of target/source patterns* $(\mathit{tar}_i, \mathit{src}_{i+1})$ *over a number of rules* b_i $(1 \leq i \leq k{-}1)$*,* $\mathit{seq} = \mathit{src}_1 \Rightarrow_{b_1} (\mathit{tar}_1, \mathit{src}_2) \Rightarrow_{b_2} \ldots \Rightarrow_{b_k} \mathit{tar}_k$ *is a* k-sequence of s/t-patterns.

$$\begin{tikzcd}
\mathit{src}_1 \triangleright L_1 & K_1 \arrow[l, hook'] \arrow[r, hook] & R_1 \arrow[dr, hook, "e_{R_1}"'] & & L_2 \arrow[dl, hook', "e_{L_2}"] & \cdots & R_k \triangleleft \mathit{tar}_k \\
& & & E_1 \triangleleft \mathit{ac}_{E_1} & & \cdots &
\end{tikzcd}$$

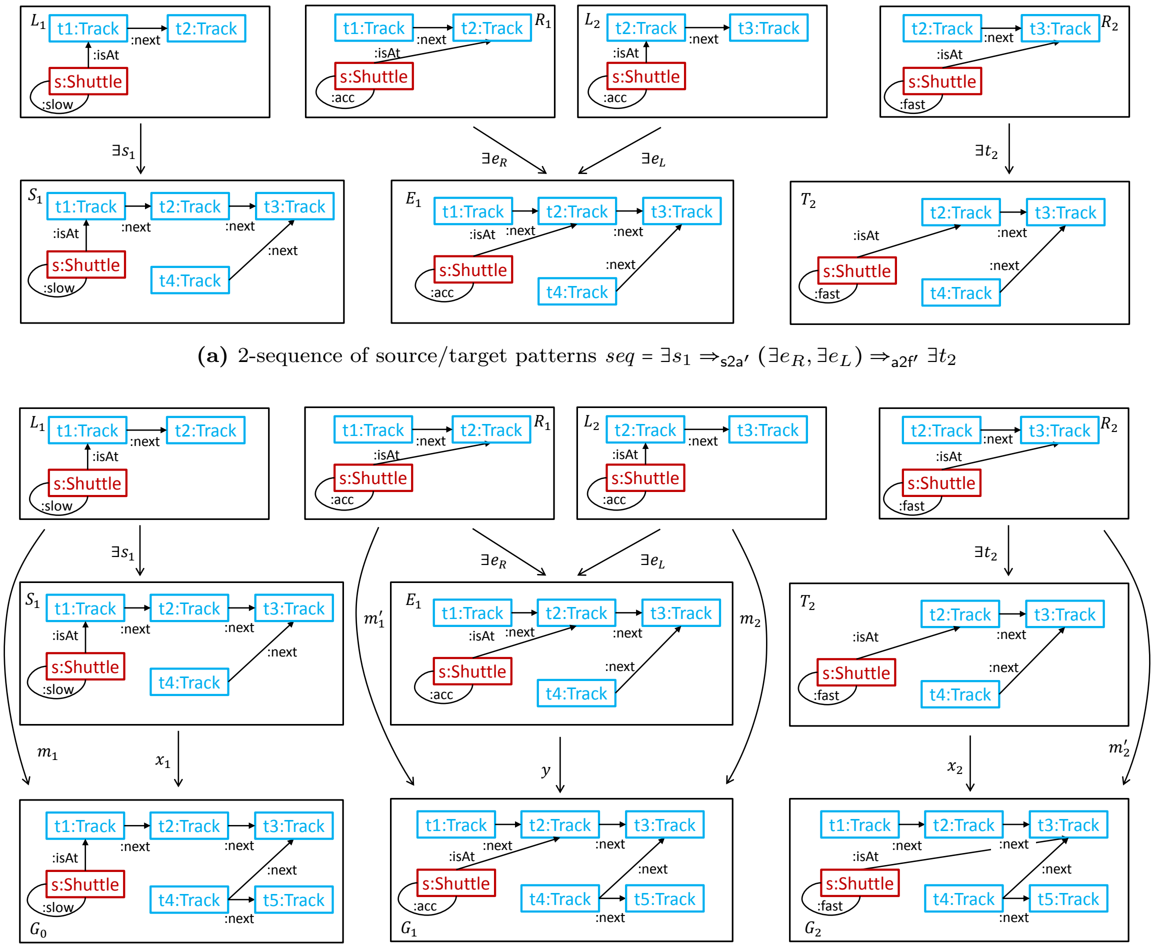
Satisfiability *of k-sequences of s/t-patterns is defined as follows:*

Given a sequence of transformations (of length k) $\mathit{trans} = G_0 \Rightarrow_{c_1, m_1, m_1'} \ldots \Rightarrow_{c_k, m_k, m_k'} G_k$ *and a k-sequence of s/t-patterns* $\mathit{seq} = \mathit{src}_1 \Rightarrow_{b_1} (\mathit{tar}_1, \mathit{src}_2) \Rightarrow_{b_2} \ldots \Rightarrow_{b_k} \mathit{tar}_k$*, trans satisfies seq, denoted as* $\mathit{trans} \vDash \mathit{seq}$*, if, for all i with* $1 \leq i \leq k$*,* $c_i = b_i$*,* $m_i \vDash \mathit{src}_i$*,* $m_i' \vDash \mathit{tar}_i$ *and, in particular, for all i with* $1 \leq i \leq k-1$*,* $(m_i', m_{i+1}) \vDash (\mathit{tar}_i, \mathit{src}_{i+1})$*.*

$$\begin{tikzcd}
\mathit{src}_1 \triangleright L_1 \arrow[dd, hook, "m_1 \vDash \mathit{src}_1"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[dd, hook] & R_1 \arrow[dr, hook, "e_{R_1}"'] \arrow[ddr, hook, bend right, "m_1'"'] \arrow[ddr, phantom, "="] & & L_2 \arrow[dl, hook', "e_{L_2}"] \arrow[ddl, hook', bend left, "m_2"] \arrow[ddl, phantom, "="] & \cdots & R_k \triangleleft \mathit{tar}_k \arrow[dd, hook, "m_k' \vDash \mathit{tar}_k"] \\
& & & E_1 \triangleleft \mathit{ac}_{E_1} \arrow[d, hook, "y_1 \vDash \mathit{ac}_{E_1}"] & & \cdots & \\
G_0 & D_1 \arrow[l, hook'] \arrow[rr, hook] & & G_1 & & \cdots & G_k
\end{tikzcd}$$

Two k-sequences of s/t-patterns seq, seq′ are equivalent *(*$\mathit{seq} \equiv \mathit{seq}'$*), if for all transformation sequences trans, it holds that* $\mathit{trans} \vDash \mathit{seq} \Leftrightarrow \mathit{trans} \vDash \mathit{seq}'$*.*

Example 5.12 (s/t-pattern sequence)**.** Figure 5.11(a) depicts an example s/t-pattern sequence of length 2 and of the form $\mathit{seq} = \mathit{src}_1 \Rightarrow_{\mathsf{s2a'}} (\mathit{tar}_1, \mathit{src}_2) \Rightarrow_{\mathsf{a2f'}} \mathit{tar}_2$ with $\mathit{src}_1 = \exists s_1$, $\mathit{tar}_1 = \exists e_R$,



(a) 2-sequence of source/target patterns $seq = \exists s_1 \Rightarrow_{\mathsf{s2a'}} (\exists e_R, \exists e_L) \Rightarrow_{\mathsf{a2f'}} \exists t_2$

(b) Transformation sequence $trans = G_0 \Rightarrow_{\mathsf{s2a'}, m_1, m_1'} G_1 \Rightarrow_{\mathsf{a2f'}, m_2, m_2'} G_2$ with $trans \models seq$

Figure 5.11. – Example s/t-pattern sequence and satisfying transformation sequence

$src_2 = \exists e_L$, and $tar_2 = \exists t_2$. In particular, the sequence describes subsequent application of s2a and a2f in the context of a switch (t3) ahead of the moving (and accelerating) shuttle (s).

Figure 5.11(b) then shows a transformation sequence $trans = G_0 \Rightarrow_{\mathsf{s2a}',m_1,m_1'} G_1 \Rightarrow_{\mathsf{a2f}',m_2,m_2'} G_1$. As depicted, there exist injective morphisms $x_1 : S_1 \hookrightarrow G_0$, $x_2 : T_2 \hookrightarrow G_2$, and $y : E_1 \hookrightarrow G_1$ such that

- $m_1 = x_1 \circ s_1$, implying $m_1 \models src_1$,
- $m_2' = x_2 \circ t_2$, implying $m_2' \models tar_2$, and
- $m_1' = y \circ e_R$ and $m_2 = y \circ e_L$, implying $(m_1', m_2) \models (tar_1, src_2)$.

Thus, *trans* satisfies *seq* and is one example of infinitely many transformation sequences represented by (i.e. satisfying) *seq*.

Note that there is a subtle difference between S_1 and E_1 on the one hand and T_2 on the other hand: T_2 misses the track t1. This is immaterial for any specific transformation sequence satisfying *seq*. Since the track is present in S_1 and E_1 and cannot be deleted by either s2a′ or a2f′, it will also be present in the transformation sequence's last graph (here: G_2). However, it is relevant for the *s*/*t*-pattern sequence (the symbolic encoding) because $tar = \exists t_2$ carries less information than src_1, tar_1, and src_2. While the definition of *s*/*t*-pattern sequences (Definition D.2 (p. 82)) does not require all patterns in a sequence to carry equivalent information – or even to be free of contradictions – this is a desirable property in sequences constructed for our verification and will be discussed later in more detail.

Furthermore, note that $seq_1 = \exists s_1 \Rightarrow_{\mathsf{s2a}'} \exists e_R$ and $seq_2 = \exists e_L \Rightarrow_{\mathsf{a2f}'} \exists t_2$ are also valid 1-sequences of *s*/*t*-patterns. Also, $trans_1 = G_0 \Rightarrow_{\mathsf{s2a}',m_1,m_1'} G_1$ and $trans_2 = G_1 \Rightarrow_{\mathsf{a2f}',m_2,m_2'} G_2$ satisfy seq_1 and seq_2, respectively.

Finally, considering our verification question and the definition of *k*-inductive invariants, we want to represent and analyze sequences possibly violating the intended *k*-inductive invariant. Following Lemma 5.2, we will want to construct *s*/*t*-pattern sequences representing transformation sequences that lead to a violation of our safety property – i.e. to a shuttle driving on a switch in speed mode fast, which is the case in G_2 of the above transformation sequence. Hence, as mentioned before, *trans* is already a counterexample for our safety property being a 2-inductive invariant for GTS' under $\mathcal{H}$, with *seq* its symbolic representation. △

This concludes the introduction of our symbolic encoding to represent and analyze transformation sequences (of possibly infinite number) in a finite fashion. The following section will focus on the construction of specific *s*/*t*-pattern sequences for our verification approach.

5.2. Construction of Pattern Sequences

In order to use *s*/*t*-pattern sequences to verify *k*-inductive invaiants and investigate $k-1$-bounded state spaces, we recall Lemmas 5.2 (p. 75) and 5.3 (p. 75) established earlier.

For Lemma 5.2 (p. 75), we attempt to prove the absence of transformation sequences $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ where G_k violates $\mathcal{F}$ ($G_k \not\models \mathcal{F}$, or $G_k \models \neg\mathcal{F}$) and satisfies $\mathcal{H}$ while all other graphs satisfy $\mathcal{F}$ and $\mathcal{H}$. We will apply this lemma without explicitly investigating all transformation sequences by finding a finite set of *s*/*t*-pattern sequences representing (being satisfied by) all transformation sequences fulfilling the condition above. If that set is empty or if all remaining *s*/*t*-pattern sequences are contradictory or will be discarded for other reasons, the safety property $\mathcal{F}$ is a *k*-inductive invariant.

A similar condition is required for proof of validity of a graph constraint $\mathcal{F}$ in the $k-1$-bounded state spaces of induced graph grammars (Lemma 5.3 (p. 75)). For the verification *k*-inductive invariants, we search for counterexamples where the rightmost graph of encoded transformation sequences satisfies one constraint ($\neg\mathcal{F}\wedge\mathcal{H}$) and all other graphs satisfy another ($\mathcal{F}\wedge\mathcal{H}$). For the analysis of $k-1$-bounded state spaces, the rightmost graph of encoded sequences should similarly satisfy $\neg\mathcal{F}\wedge\mathcal{H}$ while all other graphs should satisfy $\mathcal{H}$. Furthermore, the leftmost graph G_0 needs to satisfy the start configuration constraint $\mathcal{S}$.

To create the required s/t-pattern sequences, we introduce a generic construction applicable in both cases. In particular, we create s/t-pattern sequences for transformation sequences where the rightmost graph satisfies one generic graph constraint ($\mathcal{C}_1$) and all other graphs satisfy another ($\mathcal{C}_2$). Then, by applying the construction with the parameters $\mathcal{C}_1 = \neg\mathcal{F}\wedge\mathcal{H}$ and $\mathcal{C}_2 = \mathcal{F}\wedge\mathcal{H}$, we encode counterexamples for k-inductive invariants. For the analysis of $k-1$-bounded state spaces, we set $\mathcal{C}_1 = \neg\mathcal{F}\wedge\mathcal{H}$ and $\mathcal{C}_2 = \mathcal{H}$ – and take $\mathcal{S}$ into consideration when analyzing the resulting s/t-pattern sequences. Both cases will be explained in detail in Sections 5.3 and 5.4.

Theorem T.1g (construction of s/t-pattern sequences)**.** *There is a construction* Seq^g_k *such that for graph constraints* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*, sets of graph rules* $\mathcal{R}$*, and* $k \geq 1$, $\mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ *is a set of k-sequences of s/t-patterns such that:*

1. *For all transformation sequences trans* $= G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ *to* $\mathcal{R}$ *and of length k leading to* $\mathcal{C}_1$ *such that* $G_i \vDash \mathcal{C}_2$ *for* $0 \leq i \leq k-1$*, there exists a seq* $\in \mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ *such that trans* $\vDash$ *seq.*
2. *Given a seq* $\in \mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$*, for every transformation sequence trans* $= G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ *with trans* $\vDash$ *seq, trans leads to* $\mathcal{C}_1$ *and we have* $G_i \vDash \mathcal{C}_2$ *for* $0 \leq i \leq k-1$.

Construction. Seq^g_k *is inductively constructed as follows, starting with* Seq^g_1 *(left figure), which consists of five steps SC₁-1 to SC₁-5:*

SC₁-1: For each rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$*,* $tar_b = \mathrm{Shift}(i_R, \mathcal{C}_1)$ *is a target pattern over R.*

SC₁-2: For each such target pattern tar_b*,* $src'_b = \mathrm{L}(b, tar_b)$ *is a source pattern over L.*

SC₁-3: For each such source pattern src'_b*,* $src_b = src'_b \wedge ac_L \wedge \mathrm{Appl}(b) \wedge \mathrm{Shift}(i_L, \mathcal{C}_2)$ *is a source pattern over L.*

SC₁-4: For each such pair src_b *and* tar_b *of a source and a target pattern,* $src_b \Rightarrow_b tar_b$ *is a 1-sequence of s/t-patterns.*

SC₁-5: Finally, we define $\mathrm{Seq}^g_1(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2) = \{src_b \Rightarrow_b tar_b \mid b \in \mathcal{R}\}$ *as the set of these sequences.*

$$\begin{tikzcd}
ac_L \triangleright L & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R & \varnothing \arrow[l, hook', "i_R"'] \arrow[lll, bend right, "i_L"'] \\
src_b \triangledown & & tar_b \triangledown & \mathcal{C}_1 \triangledown \ \mathcal{C}_2 \triangleleft
\end{tikzcd}$$

$$\begin{tikzcd}
ac_L \triangleright L & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[dr, "e_R"] & \mathcal{C}_2 \triangleright \varnothing \arrow[lll, bend right, "i_L"'] & src_1 \triangleright L_1 \arrow[dl, "e_L"'] \\
src_{b,E} \triangledown & & & ac_{b,E} \triangleright E &
\end{tikzcd}$$

Given $\mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$*, we construct* $\mathrm{Seq}^g_{k+1}(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ *as follows (right figure) by six steps SC$_k$-1 to SC$_k$-5.*

SC$_k$-1: For each sequence seq $= src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k \in \mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ *with* src_1 *being a source pattern over a left rule side* L_1*, each* $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$*, and each graph E and pair of injective and jointly surjective morphisms* $(e_R : R \hookrightarrow E, e_L : L_1 \hookrightarrow E)$*,* $tar_{b,E} = \exists(e_R, ac_{b,E})$ *with* $ac_{b,E} = \mathrm{Shift}(e_L, src_1))$ *is a target pattern over R.*

SC$_k$-1$^+$: For each such target pattern $tar_{b,E}$*,* $src^+_{1,E} = \exists(e_L, ac_{b,E})$ *is a source pattern over* L_1 *and* $(tar_{b,E}, src^+_{1,E})$ *is a target/source pattern over* (b, b_1).

SC$_k$-2: For each such target pattern $tar_{b,E}$*,* $src'_{b,E} = \mathrm{L}(b, tar_{b,E})$ *is a source pattern over L.*

SC_k-3: For each such source pattern $src'_{b,E}$, $src_{b,E} = src'_{b,E} \wedge ac_L \wedge \mathrm{Appl}(b) \wedge \mathrm{Shift}(i_L, \mathcal{C}_2)$ is a source pattern over L_u.

SC_k-4: For each such pair $src_{b,E}$ and $tar_{b,E}$ of a source and a target pattern, $src_{b,E} \Rightarrow_b (tar_{b,E}, src^+_{1,E}) \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$ is a k+1-sequence of s/t-patterns.

SC_k-5: Finally, we define $\mathrm{Seq}^g_{k+1}(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2) = \{src_{b,E} \Rightarrow_b (tar_{b,E}, src^+_{1,E}) \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k \mid b \in \mathcal{R} \wedge seq \in \mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ and E as above$\}$ as the set of these sequences.

Furthermore, we define $\mathrm{SEQ}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2) = \bigcup_{1 \le i \le k} \mathrm{Seq}^g_i(\mathcal{R}, C_1, \mathcal{C}_2)$.

Proof. *1.* First, we will prove (1) by induction:

$$\begin{tikzcd}
ac_{L_1} \triangleright L_1 \arrow[dd, bend right, "m_1"'] \arrow[dd, phantom, "src_1 \triangleleft"] & K_1 \arrow[l, "l"'] \arrow[r, hook, "r"] \arrow[dd] & R_1 \arrow[dd, "m'_1"'] \arrow[dd, phantom, "tar_1 \triangledown"] & \emptyset \triangleleft \mathcal{C}_2 \arrow[l, hook, "i_{R_1}"'] \arrow[lll, bend right, "i_{L_1}"'] \arrow[ddl, bend left, "i_{G_1} \models \mathcal{C}_1"] \arrow[ddl, phantom, "\mathcal{C}_1 \triangledown \; ="'] \\
 & & & \\
G_0 & K'_1 \arrow[l] \arrow[r, hook] & G_1 &
\end{tikzcd}$$

Base case. Consider an arbitrary transformation sequence $trans = G_0 \Rightarrow_{b_1, m_1, m'_1} G_1$ to $\mathcal{R}$ with $b_1 = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), ac_{L_1}, \mathrm{true} \rangle \in \mathcal{R}$ such that $G_1 \models \mathcal{C}_1$ and $G_0 \models C_2$. Then, we have to show the existence of a s/t-pattern sequence $seq \in \mathrm{Seq}^g_1(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ such that $trans \models seq$.

By construction, there is an s/t-pattern sequence $seq \in \mathrm{Seq}^g_1(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ with $seq = src_1 \Rightarrow_{b_1} tar_1$ and $tar_1 = \mathrm{Shift}(i_{R_1}, \mathcal{C}_1)$, $src_1 = src'_1 \wedge ac_{L_1} \wedge \mathrm{Appl}(b_1) \wedge \mathrm{Shift}(i_{L_1}, \mathcal{C}_2)$, and $src'_1 = \mathrm{L}(b_1, tar_1)$.

By precondition, we have an injective morphism $i_{G_1} : \emptyset \hookrightarrow G_1$ such that $i_{G_1} \models \mathcal{C}_1$. With $G_0 \Rightarrow_{b_1, m_1, m'_1} G_1$ and the comatch $m'_1 : R_1 \hookrightarrow G_1$ in particular, we have $i_{G_1} = m'_1 \circ i_{R_1}$. Given $tar_1 = \mathrm{Shift}(i_{R_1}, \mathcal{C}_1)$ and by the Shift-construction, we have $m'_1 \models tar_1$.

Considering the source pattern $src'_1 = \mathrm{L}(b_1, tar_1)$, $G_0 \Rightarrow_{b_1, m_1, m'_1} G_1$, and the L-construction, we get $m_1 \models src'_1$. For the source pattern $src_1 = src'_1 \wedge ac_{L_1} \wedge \mathrm{Appl}(b_1) \wedge \mathrm{Shift}(i_{L_1}, \mathcal{C}_2)$ and given the transformation $G_0 \Rightarrow_{b_1, m_1, m'_1} G_1$, we have $m_1 \models ac_{L_1}$ and $m_1 \models \mathrm{Appl}(b_1)$. Since $i_{G_0} \models \mathcal{C}_2$ (by precondition), with $i_{G_0} = m_1 \circ i_{L_1}$, and by the Shift-construction, we have $m_1 \models \mathrm{Shift}(i_{L_1}, \mathcal{C}_2)$. Hence, we have $m_1 \models src_1$.

Finally, $m_1 \models src_1$ and $m'_1 \models tar_1$ imply $trans \models seq$.

$$\begin{tikzcd}
ac_L \triangleright L \arrow[dd, bend right, "m"'] \arrow[dd, phantom, "src \triangleleft"] & K \arrow[l, "l"'] \arrow[r, hook, "r"] \arrow[dd] & R \arrow[dr, "e_R"] \arrow[ddr, "m'"'] \arrow[ddr, phantom, "tar \triangledown"] & \mathcal{C}_1 \triangleright \emptyset \triangleleft \mathcal{C}_2 \arrow[lll, bend right, "i_L"'] & L_1 \triangleleft src_1 \arrow[dl, "e_L"'] \arrow[ddl, hook, "m_1"] \\
 & & & ac_E \triangleright E \arrow[d, hook, "y"] & \\
G & K' \arrow[l] \arrow[rr, hook] & & G_0 &
\end{tikzcd}$$

Inductive step. Let $\mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ be a set of s/t-pattern sequences of length k such that for each transformation sequence $trans = G_0 \Rightarrow_{b_1, m_1, m'_1} \ldots \Rightarrow_{b_k, m_k, m'_k} G_k$ that leads to C_1 and with $G_i \models C_2$ for $0 \le i \le k - 1$, there exists a $seq \in \mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ such that $trans \models seq$.

Consider a transformation sequence to $\mathcal{R}$ (of length $k + 1$) $trans' = G \Rightarrow_{b,m,m'} G_0 \Rightarrow_{b_1, m_1, m'_1} \ldots \Rightarrow_{b_k, m_k, m'_k} G_k$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle \in \mathcal{R}$ and such that $trans'$ leads to C_1 and that $G \models \mathcal{C}_2$ and $G_i \models \mathcal{C}_2$ for $0 \leq i \leq k-1$. Then, $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ is a transformation sequence leading to C_1 and with $G_i \models C_2$ for $0 \leq i \leq k-1$. By inductive hypothesis, there is an s/t-pattern sequence $seq \in \mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ with $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ such that $trans \models seq$.

Since $trans \models seq$ (and with $G_0 \Rightarrow_{b_1,m_1,m'_1} G_1$), we have a match $m_1 : L_1 \hookrightarrow G_0$ with $m_1 \models src_1$. Since there is a transformation $G \Rightarrow_{b,m,m'} G_0$, there is a comatch $m' : R \hookrightarrow G_0$. Then, given comatch and match $m' : R \hookrightarrow G_0$ and $m_1 : L_1 \hookrightarrow G_0$ and by $\mathcal{E}'$-$\mathcal{M}$-pair factorization [EGH+14], there is a graph E with a pair of jointly surjective morphisms $(e_R : R \hookrightarrow E, e_L : L_1 \hookrightarrow E)$ such that there exists an injective morphism $y : E \hookrightarrow G_0$ such that $y \circ e_R = m'$ and $y \circ e_L = m_1$. (By decomposition, e_R and e_L are injective.)

By construction, for that particular graph E and pair of injective and jointly surjective morphisms $(e_R : R \hookrightarrow E, e_L : L_1 \hookrightarrow E)$, there is an s/t-pattern sequence $seq'_E \in \mathrm{Seq}^g_{k+1}(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ with $seq' = src \Rightarrow_b (tar, src^+_1) \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ such that $(tar, src^+_1) = (\exists(e_R, ac_E), \exists(e_L, ac_E))$ is a target/source pattern with $ac_E = \mathrm{Shift}(e_L, src_1)$ and $src_E = src'_E \wedge ac_L \wedge \mathrm{Appl}(b) \wedge \mathrm{Shift}(i_L, \mathcal{C}_2)$ with $src'_E = \mathrm{L}(b, tar)$. We need to show that $trans'$ satisfies seq'.

Given $y : E \hookrightarrow G_0$ and $y \circ e_L = m_1$, $m_1 \models src_1$ (by precondition) and $ac_E = \mathrm{Shift}(e_L, src_1)$ imply $y \models ac_E$. With $y \circ e_R = m'$, the morphism pair (m', m_1) satisfies the target/source pattern (tar, src^+_1) – with $(tar, src^+_1) = (\exists(e_R, ac_E), \exists(e_L, ac_E))$.

Considering the source pattern $src' = \mathrm{L}(b, tar)$, $G \Rightarrow_{b,m,m'} G_0$, and the L-construction, we get $m \models src'$. For the source pattern $src = src' \wedge ac_L \wedge \mathrm{Appl}(b) \wedge \mathrm{Shift}(i_L, \mathcal{C}_2)$ and given the transformation $G \Rightarrow_{b,m,m'} G_0$, we have $m \models ac_L$ and $m \models \mathrm{Appl}(b)$. Since $i_G \models \mathcal{C}_2$ (by precondition), with $i_G = m \circ i_L$, and by the Shift-construction, we have $m \models \mathrm{Shift}(i_L, \mathcal{C}_2)$. Hence, we have $m \models src$.

Finally, $m \models src$, $(m', m_1) \models (tar, src^+_1)$, and $trans \models seq$ imply $trans' \models seq'$, concluding the inductive proof.

2. Second, we will prove (2) by induction:

$$\begin{tikzcd}
ac_{L_1} \triangleright L_1 \arrow[d, bend right, "m_1"', "src_1 \triangleleft"] & K_1 \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] & R_1 \arrow[dr, hook, "m'_1"', "tar_1 \triangleleft"] & \varnothing \triangleleft \mathcal{C}_2 \arrow[l, hook', "i_{R_1}"'] \arrow[lll, bend right, "i_{L_1}"'] \arrow[d, bend left, "i_{G_1}", "\mathcal{C}_1 \triangleleft"'] \arrow[dl, phantom, "="] \\
G_0 & K'_1 \arrow[l, hook'] \arrow[rr, hook] & & G_1
\end{tikzcd}$$

Base case. Consider an arbitrary s/t-pattern sequence $seq \in \mathrm{Seq}^g_1(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ with $seq = src_1 \Rightarrow_{b_1} tar_1$ and an arbitrary transformation sequence $G_0 \Rightarrow_{b_1,m_1,m'_1} G_1$ such that $trans \models seq$, implying $m'_1 \models tar_1$ and $m_1 \models src_1$. We need to show $G_1 \models \mathcal{C}_1$ and $G_0 \models C_2$.

By construction, we have $tar_1 = \mathrm{Shift}(i_{R_1}, \mathcal{C}_1)$ and by precondition, $m'_1 \models tar_1$. With $m' \circ i_{R_1} = i_{G_1}$, $m'_1 \models \mathrm{Shift}(i_{R_1}, \mathcal{C}_1)$ and by the Shift-construction, we have $i_{G_1} \models \mathcal{C}_1$, implying $G_1 \models \mathcal{C}_1$.

Given $src_1 = \mathrm{L}(b_1, tar_1) \wedge ac_{L_1} \wedge \mathrm{Appl}(b_1) \wedge \mathrm{Shift}(i_{L_1}, \mathcal{C}_2)$ and $m_1 \models src_1$, we have, in particular, $m_1 \models \mathrm{Shift}(i_{L_1}, \mathcal{C}_2)$. With $m_1 \circ i_{L_1} = i_{G_0}$ and by the Shift-construction, we have $i_{G_0} \models \mathcal{C}_2$, implying $G_0 \models \mathcal{C}_2$.

$$\begin{tikzcd}
ac_L \triangleright L \arrow[dd, bend right, "m"', "src^{\triangledown}"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[dd] & R \arrow[ddr, "m'"', "tar^{\triangledown}"] \arrow[dr, "e_R"] & \mathcal{C}_1 \triangleright \varnothing \triangleleft \mathcal{C}_2 \arrow[lll, bend right, "i_L"'] & L_1 \triangleleft src_1 \arrow[dl, "e_L"'] \arrow[ddl, "m_1"] \\
 & & & ac_E \triangleright E \arrow[d, hook, "y"] & \\
G & K' \arrow[l, hook'] \arrow[rr, hook] & & G_0 &
\end{tikzcd}$$

Inductive step. Let $\mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ be a set of s/t-pattern sequences of length k such that for each s/t-pattern sequence $seq \in \mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ and for each transformation sequence *trans* to $\mathcal{R}$ of length k and with $trans \models seq$, *trans* leads to C_1 and $G_i \models C_2$ for $0 \leq i \leq k-1$.

Consider an arbitrary s/t-pattern sequence $seq' \in \mathrm{Seq}^g_{k+1}(\mathcal{R},C_1,\mathcal{C}_2)$ with $seq' = src \Rightarrow_b (tar, src^+_1) \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ and an arbitrary transformation sequence $trans' = G \Rightarrow_{b,m,m'} G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_K,m_k,m'_k} G_k$ such that $trans' \models seq'$. We need to show that $trans'$ leads to $\mathcal{C}_1$ and that G and all G_i $(0 \leq i \leq k-1)$ satisfy $\mathcal{C}_2$.

By construction, there is an s/t-pattern sequence $seq \in \mathrm{Seq}^g_k(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2)$ with $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ such that $src^+_1 = \exists(e_L, ac_E)$ and $tar = \exists(e_R, ac_E)$ with $ac_E = \mathrm{Shift}(e_L, src_1)$ for a graph E and a pair of injective and jointly surjective morphisms $(e_L : L \hookrightarrow E, e_R : R_1 \hookrightarrow E)$. Since $trans' \models seq'$, we have $(m', m_1) \models (tar, src_1)$, implying the existence of an injective morphism $y : E \hookrightarrow G_0$ such that $y \circ e_R = m'$, $y \circ e_L = m_1$, and $y \models \mathrm{Shift}(e_L, src_1)$. Then, by the Shift-construction, we have $m_1 \models src_1$. Furthermore, considering the transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$, we get $trans \models seq$. By inductive hypothesis, *trans* leads to $\mathcal{C}_1$, i.e. $G_k \models \mathcal{C}_1$. Then, $trans'$ also leads to $\mathcal{C}_1$. Furthermore (by inductive hypothesis), all G_i with $0 \leq i \leq k-1$ satisfy $\mathcal{C}_2$.

We still need to show $G \models \mathcal{C}_2$. Since $src = \mathrm{L}(b, tar) \land ac_L \land \mathrm{Appl}(b) \land \mathrm{Shift}(i_L, \mathcal{C}_2)$ and with $m \models src$ by precondition, we have, in particular, $m \models \mathrm{Shift}(i_L, \mathcal{C}_2)$. With $m \circ i_L = i_G$ and by the Shift-construction, we have $i_G \models \mathcal{C}_2$, implying $G \models \mathcal{C}_2$, which concludes the inductive proof. □

The construction of s/t-pattern sequences comes with two important properties with respect to the constraints $\mathcal{C}_1$ and $\mathcal{C}_2$. One, all potential candidates of transformation sequences following the requirements on satisfying the constraints will be represented by one of the constructed sequences. Two, every specific transformation sequence satisfying one of the constructed s/t-pattern sequences will follow said requirements. Furthermore, since all constructions involved in the Seq-construction yield finite results, the Seq-construction has a finite result for all possible input values and a fixed k as well.

With that in mind, finding the finite set of s/t-pattern sequences representing all violations of a k-inductive invariant $\mathcal{F}$ can be reduced to constructing $\mathrm{Seq}^g_k(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. The s/t-pattern sequences representing violations of $\mathcal{F}$ under a constraint $\mathcal{H}$ can be constructed by $\mathrm{Seq}^g_k(\mathcal{R}, \neg\mathcal{F} \land \mathcal{H}, \mathcal{F} \land \mathcal{H})$. In particular, this will lead to s/t-pattern sequences whose satisfying transformation sequences will lead to $\neg\mathcal{F} \land \mathcal{H}$ while both $\mathcal{F}$ and $\mathcal{H}$ will be satisfied in earlier graphs in the sequence.

We can express the Seq-construction in a declarative fashion rather than stepwise:

$$\mathrm{Seq}^g_1(\mathcal{R},\mathcal{C}_1,\mathcal{C}_2) = \{src_b \Rightarrow_b tar_b \mid b \in \mathcal{R}\} \qquad (\mathrm{SC}_1\text{-}4/5)$$

where, given $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle$,

$$\begin{aligned} src_b &= src'_b \wedge ac_L \wedge \mathrm{Appl}(b) \wedge \mathrm{Shift}(i_L, \mathcal{C}_2), && (\mathrm{SC_1\text{-}3})\\ src'_b &= \mathrm{L}(b, tar_b), \text{ and} && (\mathrm{SC_1\text{-}2})\\ tar_b &= \mathrm{Shift}(i_R, \mathcal{C}_1). && (\mathrm{SC_1\text{-}1}) \end{aligned}$$

For the computation of $\mathrm{Seq}^g_{k+1}(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$, given $\mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$, we have:

$$\begin{aligned} \mathrm{Seq}^g_{k+1}(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2) = \{ & src_{b,E} \Rightarrow_b (tar_{b,E}, src^+_{1,E}) \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k \mid && (\mathrm{SC_k\text{-}4/5})\\ & b \in \mathcal{R} \wedge seq \in \mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2) \wedge E \in \mathcal{E}\} \end{aligned}$$

where, given $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle$, $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$, and $\mathcal{E}$ as the set of graphs with pairs of injective and jointly surjective morphisms $e_R : R \hookrightarrow E$ and $e_L : L_1 \hookrightarrow E$,

$$\begin{aligned} src_{b,E} &= src'_{b,E} \wedge ac_L \wedge \mathrm{Appl}(b) \wedge \mathrm{Shift}(i_L, \mathcal{C}_2), && (\mathrm{SC_k\text{-}3})\\ src'_{b,E} &= \mathrm{L}(b, tar_{b,E}), && (\mathrm{SC_k\text{-}2})\\ src^+_{1,E} &= \exists(e_L, ac_{b,E}), && (\mathrm{SC_k\text{-}1^+})\\ tar_{b,E} &= \exists(e_R, ac_{b,E}), \text{ and} && (\mathrm{SC_k\text{-}1})\\ ac_{b,E} &= \mathrm{Shift}(e_L, src_1). \end{aligned}$$

Regardless of length, the construction will be executed from right to left, starting with the case for length 1 (steps $\mathrm{SC_1}$-1 to $\mathrm{SC_1}$-5) and then iteratively applying the steps for prolonging the k-sequences to $k+1$ (steps $\mathrm{SC_k}$-1 to $\mathrm{SC_k}$-5). In short, the steps of the Seq-construction have the following effect:

SC$_1$-1 creates combinations of the graph constraint $\mathcal{C}_1$ with right rule sides for all rules in $\mathcal{R}$, thereby spawning one target pattern for each rule.

SC$_1$-2 creates source patterns by shifting all target patterns to the respective left rule side. Intuitively, rules are applied in reverse direction.

SC$_1$-3 adds the rules' left application condition and applicability condition to the source pattern to ensure correct rule applciation in satisfying transformation sequences. In addition, since intermediate graphs of satisfying transformation sequences have to satisfy $\mathcal{C}_2$, it is shifted to the source pattern, too.

SC$_1$-4/5 combines the pairs of source and target patterns into a number of s/t-pattern sequences of length 1.

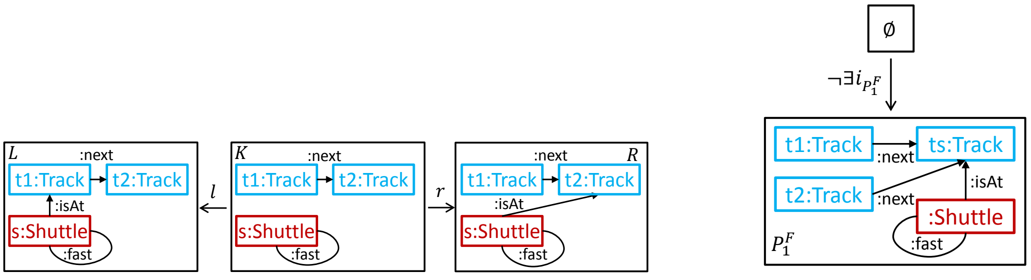
SC$_\mathrm{k}$-1 creates combinations of the leftmost source pattern and the right rule sides for all s/t-pattern sequences in $\mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ and rules in $\mathcal{R}$, thereby spawning a number of target patterns for each pair of a rule and a sequence.

SC$_\mathrm{k}$-1$^+$ makes sure there is equivalent information in the newly created target patterns and the (leftmost) source patterns used in their creation and combines them in target/source patterns.

SC$_\mathrm{k}$-2 creates source patterns by shifting all target patterns to the respective left rule side. Intuitively, rules are applied in reverse direction.

SC$_\mathrm{k}$-3 adds the rules' left application condition and applicability condition to the source pattern to ensure correct rule applciation in satisfying transformation sequences. In addition, since intermediate graphs of satisfying transformation sequences have to satisfy $\mathcal{C}_2$, it is shifted to the source pattern, too.

SC$_\mathrm{k}$-4/5 combines the constructed source patterns and target/source patterns with their corresponding sequences in $\mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ to form a number of s/t-pattern sequences of length $k+1$.



(a) Graph rule f2f′ **(b)** Safety property $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

Figure 5.12. – Graph rule f2f′ and graph constraint $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

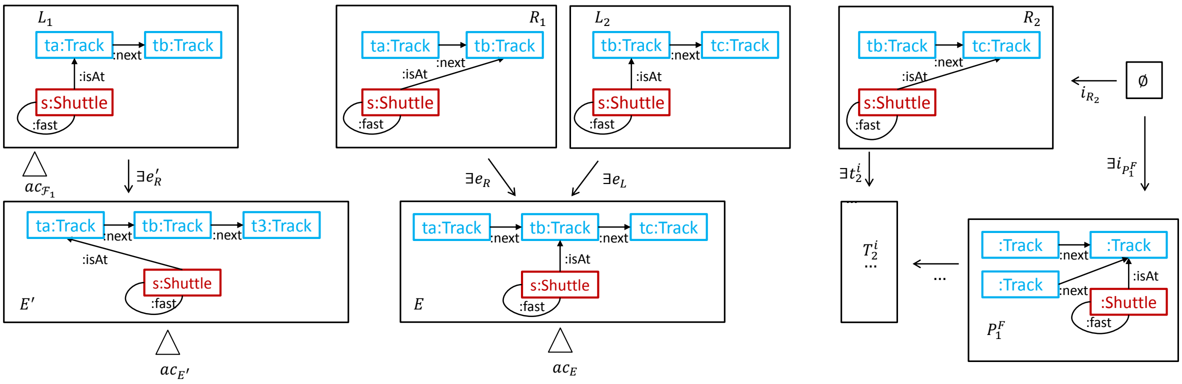


Figure 5.13. – The s/t-pattern sequence $seq_2 = src_1 \Rightarrow_{\mathsf{f2f}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$ with $seq_2 \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$

Example 5.13 (Seq-construction, example system)**.** In order to keep this example short, we will use a graph transformation system $GTS = (TG, \mathcal{R})$, where the set of rules consists only of the graph rule $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ (hence, $\mathcal{R} = \{\mathsf{f2f'}\}$). In addition, we have a safety property $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$. The rule and the graph constraint $\neg F_1$ are shown in Figures 5.12(a) and 5.12(b), respectively; they are the same as in Example 5.1 (p. 69).

With respect to the graph rule, we will distinguish between $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ and $\mathsf{f2f'} = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), \mathrm{true}, \mathrm{true}\rangle$. The former will refer to the appearance of rule f2f′ in the context of steps SC$_1$-1 to SC$_1$-5 and the latter to its appearance in the context of steps SC$_\mathrm{k}$-1 to SC$_\mathrm{k}$-5, although the rules' contents are identical (i.e. $L_1 = L_2$ and so on).

We will compute $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, which would be appropriate in order to determine whether $\mathcal{F}$ is a 2-inductive invariant for GTS under a guaranteed constraint with the trivial value true. Here, we will not consider the guaranteed constraint $\mathcal{H}$ introduced in Example 5.1 for reasons of (visual) complexity.

Figure 5.13 shows one s/t-pattern sequence (of length 2) seq_2 that is contained in $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. The construction and origin of its individual parts are explained in Examples A.2-A.10 in Appendix A; here, we will only provide a short overview.

In particular,

$$\begin{aligned} seq_2 = \quad & src_1 && \Rightarrow_{\mathsf{f2f'}} && (tar_1, src_2^+) && \Rightarrow_{\mathsf{f2f'}} && tar_2 \\ = & \exists(e'_R, ac'_E) \wedge ac_{\mathcal{F}_1} && \Rightarrow_{\mathsf{f2f'}} && (\exists(e_R, ac_E), \exists(e_L, ac_E)) && \Rightarrow_{\mathsf{f2f'}} && \bigvee_{i \in I} \exists t_2^i, \end{aligned}$$

where the steps, their computations, and the corresponding examples and figures for this particular s/t-pattern sequence are listed in Table 5.1. Roughly and intuitively, the end result

Table 5.1. – Computation steps of $seq_2 \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$

Step	Computation	Figure	Example
SC$_1$-1	$tar_2 = \bigvee_{i\in I} \exists t_2^i = \mathrm{Shift}(i_{R_2}, F_1)$	A.3	A.2
SC$_1$-2	$src_2' = \bigvee_{i\in I} \exists s_2^i = \mathrm{L}(\mathsf{f2f'}, tar_2)$	A.4	A.3
SC$_1$-3	$src_2 = \bigvee_{i\in I} \exists s_2^i \wedge ac_{\mathcal{F}_2}$ with $ac_{\mathcal{F}_2} = \mathrm{Shift}(i_{L_2}, \mathcal{F})$	A.5	A.4
SC$_1$-4/5	$seq_1 = src_2 \Rightarrow_{\mathsf{f2f'}} tar_2$	A.6	A.5
SC$_\mathrm{k}$-1	$tar_1 = \exists(e_R, ac_E)$ with $ac_E = \mathrm{Shift}(e_L, src_2)$	A.8	A.6
SC$_\mathrm{k}$-1$^+$	$src_2^+ = \exists(e_L, ac_E)$	–	A.7
SC$_\mathrm{k}$-2	$src_1' = \exists(e_R', ac_{E'}) = \mathrm{L}(\mathsf{f2f'}, tar_1)$	A.9	A.8
SC$_\mathrm{k}$-3	$src_1 = \exists(e_R', ac_{E'}) \wedge ac_{\mathcal{F}_1}$ with $ac_{\mathcal{F}_1} = \mathrm{Shift}(i_{L_1}, \mathcal{F})$	A.10	A.9
SC$_\mathrm{k}$-4/5	$seq_2 = src_1 \Rightarrow_{\mathsf{f2f'}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f'}} tar_2$	5.14, A.11	A.10

is this: the disjunction over the existential conditions $\exists t_2^i$ describes all possibilities where the application of rule $\mathsf{f2f'}$ has led to a shuttle driving fast on a switch; then, reverse applications of the rule via the L-construction determine the situation before those rule applications. The sequences in $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ differ by their overlappings of the first and second rule (which are both $\mathsf{f2f'}$). In seq_2, that overlapping is represented by the morphism pair (e_R, e_L) and the graph E.

In full detail, we have:

$$\begin{aligned}
seq_2 = \;& \exists(e_R', \bigvee_{j\in J}\bigvee_{a\in A_j} \exists s_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists s_{1*}^{jd}) \wedge \bigwedge_{j\in J} \neg\exists c_1^j \\
\Rightarrow_{\mathsf{f2f'}} & (\exists(e_R, \bigvee_{j\in J}\bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd}), \\
& \quad \exists(e_L, \bigvee_{j\in J}\bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd})) \\
\Rightarrow_{\mathsf{f2f'}} & \qquad \bigvee_{i\in I} \exists t_2^i,
\end{aligned}$$

with some of the elements and connections depicted in Figure 5.14. As a counterexample for human inspection, this is not very helpful: at some point during the construction, the amount of conditions and graphs exceed what can be seen and understood in reasonable time and effort by a human viewer. Still, the graph E (and its alternatives given other possible morphism pairs (e_R, e_L)) provides an indication of the interaction of subsequently applied rules in the s/t-pattern sequence at hand. △

In the following paragraphs, we will delve deeper into the details of the Seq-construction on a general level. Detailed examples are listed in Appendix A.

Step SC$_1$-1: For each rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$, $tar_b = \mathrm{Shift}(i_R, \mathcal{C}_1)$ is a target pattern over R.

Given a rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle$, $\mathrm{Shift}(i_R, \mathcal{C}_1)$ transfers the constraint $\mathcal{C}_1$ to the context of the right rule side R in order to create the sequences' rightmost target pattern. Since the construction of s/t-pattern sequences (and hence, the propagation of information) is inductively executed from right to left, the resulting target pattern $tar = \mathrm{Shift}(i_R, \mathcal{C}_1)$ will not be changed later in the construction, regardless of the sequences' length. The result is encoding satisfiability of $\mathcal{C}_1$ in the context of the result of a rule application (i.e. occurrence of the right rule side).

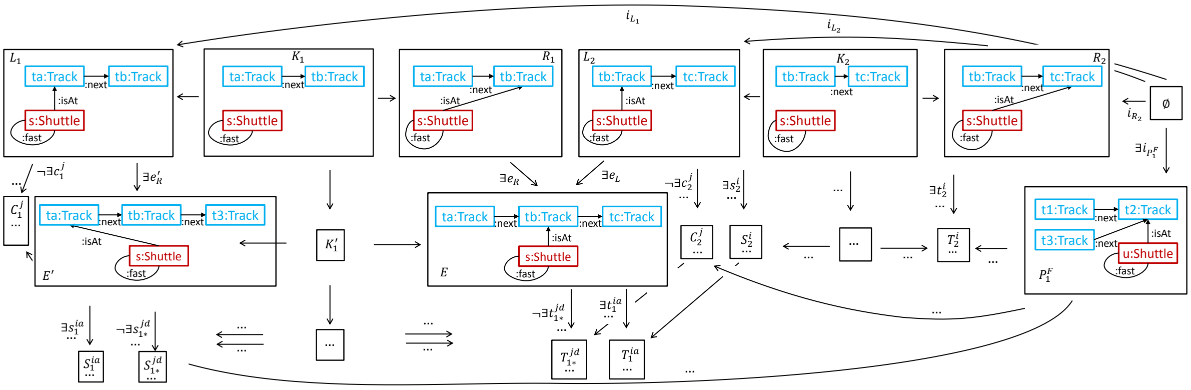


Figure 5.14. – Steps SC$_\text{k}$-4/5: $seq_2 = src_1 \Rightarrow_{\text{f2f}'} (tar_1, src_2^+) \Rightarrow_{\text{f2f}'} tar_2$ with $seq_2 \in \text{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$

The Shift-construction guarantees that, given any rule application's comatch $m' : R \hookrightarrow G$ for any graph G, m' will satisfy tar if and only if G satisfies the constraint $\mathcal{C}_1$. This is the basis for fulfilling the $\mathcal{C}_1$ part of the Seq-construction's properties. By definition, a satisfying transformation sequence means $m' \vDash tar$, which implies satisfaction of $\mathcal{C}_1$ by G (the sequence's rightmost graph); conversely, $G \vDash \mathcal{C}_1$ and the existence of the respective comatch will imply $m' \vDash tar$, guaranteeing that there is a representation for the respective transformation sequence. Since the construction needs to take all rules in a rule set into account, Shift$(i_R, \mathcal{C}_1)$ has to be computed for all rules (and right rule sides R, respectively); there will be one target pattern per rule.

Since we employ s/t-pattern sequences in order to represent possibly infinitely many cases in a finite fashion, it is important to highlight that Shift always yields a finite result by construction – specifically, an application condition, which are finite by definition.

Step SC$_1$-2: For each such target pattern tar_b, $src_b' = \text{L}(b, tar_b)$ is a source pattern over L.

After having established target patterns of the form $tar = \text{Shift}(i_R, \mathcal{C}_1)$ in the previous step, $src' = \text{L}(b, tar)$ will transform each target pattern into a source pattern over the respective rule's left side. Intuitively speaking, given a target pattern, the rule will be applied in reverse direction to determine the (symbolic) state before rule application. In particular, by the L-construction, any rule application $G' \Rightarrow_{b,m,m'} G$ will imply the equivalence of $m \vDash src'$ and $m' \vDash tar$: if the source pattern is satisfied before rule application, the target pattern will be satisfied after rule application and the resulting graph will satisfy $\mathcal{C}_1$ (see step SC$_1$-1).

Again, the result of this construction is finite. Given a number of target patterns equal to the number of graph rules considered, the result of this step is an equal number of corresponding source patterns.

Step SC$_1$-3: For each such source pattern src_b', $src_b = src_b' \wedge ac_L \wedge \text{Appl}(b) \wedge \text{Shift}(i_L, \mathcal{C}_2)$ is a source pattern over L.

This step ensures the applicability of the rule in question for any transformation sequence satisfying the s/t-pattern sequence under construction and implements the requirement that intermediate graphs of a satisfying transformation sequence satisfy $\mathcal{C}_2$. Conjunctively joining the respective left application condition ac_L and rule applicability condition Appl(b) to each source pattern achieves the former: any potential match $m : L \hookrightarrow G'$ to any graph G' that satisfies ac_L and Appl(b) will imply the existence of a transformation $G' \Rightarrow_{b,m,m'} G$ (by

Lemma 2.30 (p. 37)). $\mathrm{Shift}(i_L, \mathcal{C}_2)$, on the other hand, makes sure that any graph (except the last) appearing in a satisfying transformation sequence also satisfies $\mathcal{C}_2$ as required.

Step SC$_1$-4: For each such pair src_b and tar_b of a source and a target pattern, $src_b \Rightarrow_b tar_b$ is a 1-sequence of s/t-patterns.

Step SC$_1$-5: Finally, we define $\mathrm{Seq}_1^g(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2) = \{src_b \Rightarrow_b tar_b \mid b \in \mathcal{R}\}$ as the set of these sequences.

These steps put the computed target patterns and corresponding source patterns together to form a number of 1-sequences of source/target patterns. By construction, that number is equal to the number of rules in the set of graph rules.

By Theorem T.1g (p. 85), any transformation sequence $trans = G_1 \Rightarrow_{\mathcal{R}} G_2$ with $G_2 \models \mathcal{C}_1$ and $G_1 \models \mathcal{C}_2$ has a representing s/t-pattern sequence seq in $\mathrm{Seq}_1(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$, i.e. $trans \models seq$. Also, given a transformation sequence $trans = G_1 \Rightarrow_{\mathcal{R}} G_2$ that satisfies an s/t-pattern sequence $seq \in \mathrm{Seq}_1(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$, we know that $G_2 \models \mathcal{C}_1$ and $G_1 \models \mathcal{C}_2$.

Step SC$_k$-1: For each sequence $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k \in \mathrm{Seq}_k^g(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ with src_1 being a source pattern over a left rule side L_1, each $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle \in \mathcal{R}$, and each graph E and pair of injective and jointly surjective morphisms $(e_R : R \hookrightarrow E, e_L : L_1 \hookrightarrow E)$, $tar_{b,E} = \exists(e_R, ac_{b,E})$ with $ac_{b,E} = \mathrm{Shift}(e_L, src_1))$ is a target pattern over R.

The following steps take all constructed s/t-pattern sequences of length k and prolongs them to create the required sequences of length $k+1$. Intuitively, this step applies the same idea as step SC$_1$-1, but considers the leftmost source patterns of the sequences of length k instead of the constraint $\mathcal{C}_1$. In other words, the target pattern resulting from this step encode all situations where a rule application leads to the start of a s/t-pattern sequence of length k that fulfills the required properties with respect to the constraints $\mathcal{C}_1$ and $\mathcal{C}_2$.

Given a source pattern src_1 and a rule, a disjunction of the resulting target patterns for all possible morphisms e_R would be the result of shifting said source pattern to the respective right rule side R: $\bigvee_{e_R, E \in \dots} \exists(e_R, ac_E) = \mathrm{Shift}(i_R, src_1)$. Here, we have chosen to create a new target pattern and, in step SC$_k$-4/5, a new s/t-pattern sequence from each such existential condition $\exists(e_R, ac_E)$. As before, all rules need to be considered. Also, there is a finite number of graphs E and injective and jointly surjective morphism pairs and hence, there is a finite number of resulting target patterns.

Step SC$_k$-1$^+$: For each such target pattern $tar_{b,E}$, $src_{1,E}^+ = \exists(e_L, ac_{b,E})$ is a source pattern over L_1 and $(tar_{b,E}, src_{1,E}^+)$ is a target/source pattern over (b, b_1).

By the definition of s/t-pattern sequences (Definition D.2 (p. 82)), a s/t-pattern sequence of length 2 (or longer) is not merely a list of two (or more) 1-sequences. Rather, they have to be glued together by combining a target and a subsequent source pattern to form a target/source pattern. In particular, the target patterns created in the previous steps need to be connected to the s/t-pattern sequence used in their construction by creating target/source patterns. Since target/source patterns consist of two existential conditions with the same codomain and a nested condition and since the created target patterns have the form $tar = \exists(e_R, ac_E)$, we have to shift the corresponding source patterns, which are application conditions over a left rule side L_1, over the morphism e_L to E: $src_{b,E}^+ = \exists(e_L, ac_E)$, where, in particular, $ac_E = \mathrm{Shift}(e_L, src_1)$. Then, the target/source pattern is of the form $(tar_{b,E}, src_{1,E}^+) = (\exists(e_R, ac_E), \exists(e_L, ac_E))$.

With respect to notation, note that src^+ usually denotes the extension of a source pattern src to a source pattern src^+ that is used as part of a target/source pattern. Conversely, src^- refers to an 'underlying' source pattern that has been extended to a source pattern src appearing in a target/source pattern.

Step $\mathsf{SC_k}$-2: For each such target pattern $tar_{b,E}$, $src'_{b,E} = \mathrm{L}(b, tar_{b,E})$ is a source pattern over L.

As before (step SC$_1$-2), this step transfers all newly constructed target patterns over the rule in reverse direction to create corresponding source pattern over the respective rules' left sides.

Step $\mathsf{SC_k}$-3: For each such source pattern $src'_{b,E}$, $src_{b,E} = src'_{b,E} \wedge ac_L \wedge \mathrm{Appl}(b) \wedge \mathrm{Shift}(i_L, \mathcal{C}_2)$ is a source pattern over L_u.

Similar to step SC$_1$-3, we add the rule's left application condition, the rule applicability condition, and the constraint $\mathcal{C}_2$ transferred to the context of the left rule side to the source pattern.

Step $\mathsf{SC_k}$-4: For each such pair $src_{b,E}$ and $tar_{b,E}$ of a source and a target pattern, $src_{b,E} \Rightarrow_b (tar_{b,E}, src^+_{1,E}) \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ is a k+1-sequence of s/t-patterns.

Step $\mathsf{SC_k}$-5: Finally, we define $\mathrm{Seq}^g_{k+1}(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2) = \{src_{b,E} \Rightarrow_b (tar_{b,E}, src^+_{1,E}) \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k \mid b \in \mathcal{R} \wedge seq \in \mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ and E as above$\}$ as the set of these sequences.

In these steps, we combine the newly created and corresponding target/source patterns and source patterns in order to create all s/t-pattern sequence of length $k+1$. The target/source patterns (step SC$_\mathrm{k}$-1$^+$) connect the created target patterns (step SC$_\mathrm{k}$-1) to the leftmost source patterns computed in the previous iteration (i.e. for length k) of the Seq-construction. The result is a finite set of s/t-pattern sequences of length $k+1$. From here on, steps SC$_\mathrm{k}$-1 to SC$_\mathrm{k}$-5 can be repeated until the desired length of the s/t-pattern sequences is reached.

By Theorem T.1g (p. 85), any transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ that leads to $\mathcal{C}_1$ with intermediate graphs satisfying $\mathcal{C}_2$ has a representing s/t-pattern sequence seq in $\mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$, i.e. $trans \vDash seq$. Also, given a transformation sequence $trans$ that satisfies an s/t-pattern sequence $seq \in \mathrm{Seq}_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$, we know that $trans$ leads to $\mathcal{C}_1$, while earlier graphs in the sequence satisfy $\mathcal{C}_2$.

Seq-construction and Satisfiability. While it establishes symbolic representations, the Seq-construction does not describe how to obtain concrete transformation sequences satisfying the constructed s/t-pattern sequences. Furthermore, it does not even guarantee the existence of such a transformation sequence. For instance, the results of a Seq-construction for $k = 2$ could be equivalent to false $\Rightarrow_{b_1}$ false $\Rightarrow_{b_2}$ false, which cannot have a satisfying transformation sequence. For the purpose of analysis, it is desirable to establish the existence or absence of satisfying transformation sequences. For purposes of human inspection of counterexamples to a k-inductive invariant, it is desirable to create specific transformation sequences even if their existence has already been established. The following lemma considers this aspect.

Lemma 5.14 (existence of satisfying transformation sequences). *Given an s/t-pattern sequence $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k) \in \mathrm{Seq}^g_k(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$ and a graph G_0 such that $G_0 \vDash src_{1|\varnothing}$, there is a transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ such that $trans \vDash seq$.*

$$\begin{tikzcd}
src_{1|\varnothing} \rhd \varnothing \arrow[r, bend left, "i_{L_1}"] \arrow[dr, "i_{G_0} \vDash src_{1|\varnothing}"'] & src_1 \rhd L_1 \arrow[d, hook, "m_1 \vDash src_1"] & \cdots \arrow[l, hook'] & \cdots \arrow[r, hook] & R_k \lhd tar_k \arrow[d, hook, "m'_k \vDash tar_k"] \\
 & G_0 & \cdots \arrow[l, hook'] & \cdots \arrow[r, hook] & G_k
\end{tikzcd}$$

Proof. We will show the required statement by structural induction.

Induction base. By definition of satisfiability and Lemma 2.38, $G_0 \vDash src_{1|\varnothing}$ implies the existence of an injective morphism $m_1 : L_1 \hookrightarrow G_0$ such that $m_1 \vDash src_1$. By construction of Seq_k^g, there is a source pattern src_1' such that $src_1 = src_1' \wedge ac_{L_1} \wedge \mathrm{Appl}(b_1) \wedge \mathrm{Shift}(i_{L_1}, \mathcal{C}_2)$ and $src_1' = \mathrm{L}(b_1, tar_1)$. Since $m_1 \vDash src_1$, we have $m_1 \vDash ac_{L_1} \wedge \mathrm{Appl}(b_1)$ and by Lemma 2.30, there is a graph G_1 and transformation $G_0 \Rightarrow_{b_1, m_1, m_1'} G_1$. Furthermore, $m_1 \vDash src_1'$ and, by the L-Lemma, we have $m_1' \vDash tar_1$.

$$\begin{tikzcd}
src_{1|\varnothing} \triangleright \varnothing \arrow[r, bend left, "i_{L_1}"] \arrow[dr, "i_{G_0} \vDash src_{1|\varnothing}"'] & src_1 \triangleright L_1 \arrow[d, hook, "m_1 \vDash src_1"] & K_1 \arrow[l, hook'] \arrow[r, hook] & R_1 \triangleleft tar_1 \arrow[d, hook, "m_1' \vDash tar_1"] & \cdots \\
& G_0 & K_1' \arrow[l, hook'] \arrow[r, hook] & G_1 & \cdots
\end{tikzcd}$$

Inductive step. By inductive hypothesis, given $m_i' : R_i \hookrightarrow G_i$ with $m_i' \vDash tar_i$ for an i with $1 \leq i < k$, we also have, by the Seq-construction, $tar_i = \exists(e_R, ac_E)$ and $src_{i+1} = \exists(e_L, ac_E)$ as shown below. By construction, there is an underlying source pattern src_{i+1}^- such that $ac_E = \mathrm{Shift}(e_L, src_{i+1}^-)$ and that $seq' = src_{i+1}^- \Rightarrow_{b_{i+1}} \ldots \Rightarrow_{b_k} tar_k$ is a $(k-i)$-sequence of source/target patterns such that $seq' \in \mathrm{Seq}_{k-i}^g(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$.

$$\begin{tikzcd}
\cdots & R_i \arrow[d, hook, "e_R"] \arrow[dd, hook, bend right=60, "m_i'"', "ac_E \triangleright"] & src_{i+1}^- \triangleleft & & & \\
& E \arrow[d, hook, "y"] \arrow[dr, phantom, "="] & L_{i+1} \arrow[l, hook', "e_L"] \arrow[dl, hook, "m_{i+1} \vDash src_{i+1}^-"] & K_{i+1} \arrow[l, hook'] \arrow[r, hook] & R_{i+1} \triangleleft tar_{i+1} \arrow[d, hook, "m_{i+1}' \vDash tar_{i+1}"] & \cdots \\
\cdots & G_i & & K_{i+1}' \arrow[ll, hook'] \arrow[r, hook] & G_{i+1} & \cdots
\end{tikzcd}$$

Furthermore, $m_i' \vDash tar_i$ implies the existence of an injective morphism $y : E \hookrightarrow G_i$ such that $y \vDash ac_E$. Then, there is an injective morphism $m_{i+1} = y \circ e_L$ (i.e. $m_{i+1} : L_{i+1} \hookrightarrow G_i$), implying $m_{i+1} \vDash src_{i+1}$ (via y) and hence, $(m_i', m_{i+1}) \vDash (tar_i, src_{i+1})$. With $ac_E = \mathrm{Shift}(e_L, src_{i+1}^-)$ and the Shift-lemma, $y \vDash ac_E$ implies $y \circ e_L \vDash src_{i+1}^-$ and, consequently, $m_{i+1} \vDash src_{i+1}^-$.

Since $seq' \in \mathrm{Seq}_{k-i}^g(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$, we know that $src_{i+1}^- = src_{i+1}' \wedge ac_{l_{i+1}} \wedge \mathrm{Appl}(b_{i+1}) \wedge \mathrm{Shift}(i_{L_{i+1}}, \mathcal{C}_2)$ and $src_{i+1}' = \mathrm{L}(b_{i+1}, tar_{i+1})$. Consequently, $m_{i+1} \vDash src_{i+1}^-$ implies $m_{i+1} \vDash ac_{L_{i+1}} \wedge \mathrm{Appl}(b_{i+1})$, which in turn implies the existence of a transformation $G_i \Rightarrow_{b_{i+1}, m_{i+1}, m_{i+1}'} G_{i+1}$. By the L-lemma, we have $m_{i+1}' \vDash tar_{i+1}$, concluding the inductive proof. □

The intuitive idea behind the proof is to use the s/t-pattern sequence's leftmost source pattern to instantiate a graph satisfying its reduction to a graph constraint. Given that graph, the sequence's rules can be applied subsequently in forward direction to create a transformation sequence that satisfies the s/t-pattern sequence. Applicability of the rules and existence of the individual transformations are guaranteed because of the Seq-construction's step $\mathrm{SC_k}$-3 (and $\mathrm{SC_1}$-3): there, the rules' left application condition and the rule applicability condition are conjunctively joined to the respective source pattern.

Example 5.15. Consider $seq_2 \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ as given in Example 5.13 (p. 90). Its leftmost source pattern is $src_1 = \exists(e_R', ac_{E'}) \wedge ac_{\mathcal{F}_1}$ and its reduction to a constraint is $src_{1|\varnothing} = \exists(i_{E'}, ac_{E'}) \wedge ac_{\mathcal{F}_1|\varnothing}$. Figure 5.15 shows a transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f'}, m_1, m_1'} G_1 \Rightarrow_{\mathsf{f2f'}, m_2, m_2'} G_2$. In particular, $G_0 \vDash src_{1|\varnothing}$. By Lemma 5.14, there exists a transformation

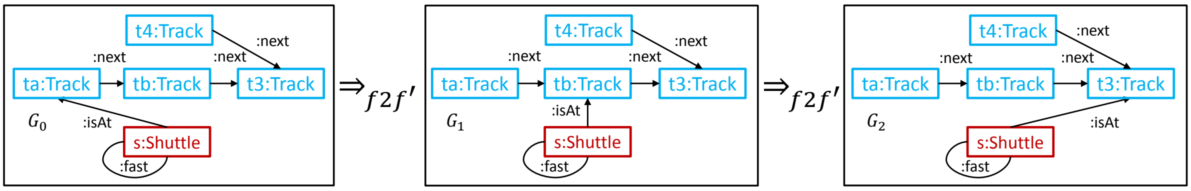


Figure 5.15. – Transformation sequence *trans* = $G_0 \Rightarrow_{\mathsf{f2f}',m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f}',m_2,m_2'} G_2$ with *trans* $\models$ seq_2

sequence that satisfies seq_2 starting from G_0 – and indeed, *trans* in Figure 5.15 is such a sequence. A more detailed version of this example is shown in Example A.11 in Appendix A. △

Note that this lemma and its proof does not provide a procedure to find a satisfying graph $G_0 \models src_{1|\varnothing}$ nor does it construct an actual satisfying transformation sequence. Indeed, although the proof explains a possible idea to construct a transformation sequence, the lemma is only concerned with proving the existence of such a sequence. Constructions to find satisfying graphs and transformation sequences have been discussed in the literature [HP09, SLO17]. Unfortunately, the general problem of finding satisfying graphs for a graph constraint is undecidable [HP09].

Only the leftmost source pattern of an s/t-pattern sequence created by the Seq-construction contains all accumulated information: all conditions are appropriately shifted or transferred by the Shift- and L-constructions. Since Lemma 5.14 (p. 94) only requires the leftmost source pattern, the incomplete information in other source and target patterns is not a problem when it comes to finding satisfying transformation sequences. However, if we were to analyze properties of the other source or target patterns (individually) without taking accumulated information into account, we may get a different result. We will consider the implications of this aspect when explaining the restricted approach to k-inductive invariant checking (Chapter 6).

With both the construction of s/t-pattern sequences and the question of satisfying transformation sequences addressed, we can move on to the core idea and main motivation: verifying a graph constraint as a k-inductive invariant for a graph transformation system under a constraint.

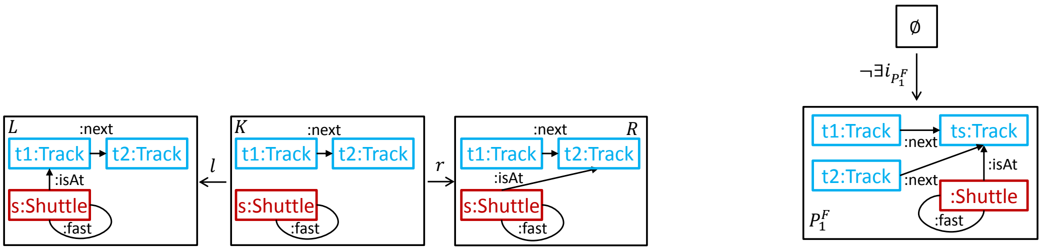
5.3. k-inductive Invariant Checking

Finding counterexamples to a k-inductive invariant under a guaranteed constraint amounts to finding transformation sequences of length k that lead to a violation of the desired invariant while all earlier graphs fulfill the invariant and all graphs in the sequence satisfy the guaranteed constraint. By Theorem T.1g (p. 85), such transformation sequences are represented by s/t-pattern sequences in $\mathrm{Seq}^g_k(\mathcal{R}, \neg\mathcal{F} \wedge H, \mathcal{F} \wedge H)$. Formally, we establish this connection in our central verification theorem for the general approach to k-inductive invariant checking:

Theorem T.2g (k-inductive invariant checking)**.** *Let* $GTS = (\mathcal{R}, TG)$ *be a typed graph transformation system and* $\mathcal{F}$ *and* $\mathcal{H}$ *be graph constraints.*

$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ if and only if, for all sequences $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$ *with* $seq \in \mathrm{Seq}^g_k(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$*, there does not exist a graph G such that* $G \models src_{1|\varnothing}$.

Proof. According to Lemma 5.2 (p. 75), we need to prove that there does not exist a transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m_1'} \ldots \Rightarrow_{b_k,m_k,m_k'} G_k$ leading to $\neg\mathcal{F} \wedge \mathcal{H}$ with $G_i \models \mathcal{F} \wedge \mathcal{H}$ for $0 \leq i \leq k-1$ if and only if the above condition holds.



(a) Graph rule f2f′

(b) $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

Figure 5.16. – Graph rule and intended 2-inductive invariant

If. Assume that $\mathcal{F}$ is not a k-inductive invariant for GTS under $\mathcal{H}$. Then, there exists a transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ that leads to $\neg\mathcal{F} \wedge \mathcal{H}$ with $G_i \vDash \mathcal{F} \wedge \mathcal{H}$ for $0 \le i \le k-1$. By Theorem T.1g (p. 85), there exists a s/t-pattern sequence of length k $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k) \in \mathrm{Seq}^g_k(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$ such that $trans \vDash seq$. Then, we have $m_1 \vDash src_1$. By Lemma 2.38 (p. 43), we get $G_0 \vDash src_{1|\varnothing}$, which is a contradiction; hence, $trans$ cannot exist and $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.

Only if. We will show this statement by contraposition. Assume the existence of an s/t-pattern sequence $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k) \in \mathrm{Seq}^g_k(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$ such that there exists a graph G_0 with $G_0 \vDash src_{1|\varnothing}$. By Lemma 5.14 (p. 94), there exists a transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ such that $trans \vDash seq$. By Theorem T.1g (p. 85), $trans$ then leads to $\neg\mathcal{F} \wedge \mathcal{H}$ and all its graphs $G_0, \dots, G_{k-1}$ satisfy $\mathcal{F} \wedge \mathcal{H}$. Hence, $\mathcal{F}$ is not a k-inductive invariant for GTS under $\mathcal{H}$. □

Intuitively, the proof combines the properties established in Theorem T.1g (p. 85) with the possibility of finding satisfying transformation sequences for an s/t-pattern sequences described by Lemma 5.14 (p. 94). After encoding all possible counterexamples (transformation sequences) in s/t-pattern sequences ($\mathrm{Seq}^g_k(\mathcal{R}, \neg F \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$), we have to find out whether there are actual transformation sequences capable of satisfying the s/t-pattern sequences and establishing counterexamples (Lemma 5.14 (p. 94)). Note that, as established by the theorem, we only have to find (or disprove the existence of) a satisfying graph $G_0 \vDash src_{1|\varnothing}$; further analysis of the corresponding transformation sequence or s/t-pattern sequence is not required.

The theorem describes an equivalence: there are no false negatives or false positives. This also shows that the information accumulated in the leftmost source pattern of an s/t-pattern sequence constructed by the Seq-construction is complete with respect to the question of finding of rejecting counterexamples. However, the problem of finding satisfying graphs for nested graph constraints is, in general, undecidable. Hence, an implementation of the procedure described by Theorem T.1g (p. 85), Lemma 5.14 (p. 94), and Theorem T.2g (p. 96) cannot be sound, complete, and terminating at once.

Example 5.16 (2-inductive invariant checking for an unsafe system)**.** This example follows and is based on Examples 5.13 (p. 90) and 5.15 (p. 95) and their detailed variants in Examples A.1-A.11. Again, we have a rule set $\mathcal{R} = \{\mathsf{f2f'}\}$ (Figure 5.16(a)) and a corresponding graph transformation system $GTS = (TG, \mathcal{R})$. Likewise, our safety property will again be the absence of a shuttle driving on a switch in speed mode fast (Figure 5.16(b)) and we intend for the constraint $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$ to be a 2-inductive invariant of GTS without further restrictions, i.e. under the trivial constraint $\mathcal{H} = \mathrm{true}$.

Then, $seq_2 = src_1 \Rightarrow_{\mathsf{f2f'}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f'}} tar_2$ (Example 5.13, Figure 5.14 (p. 92)) is an s/t-pattern sequence in $\mathrm{Seq}^g_2(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. Furthermore, G_0 (Example 5.15 (p. 95)) satisfies $src_{1|\varnothing}$; hence, $\mathcal{F}$ is not a 2-inductive invariant of GTS. The transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f'}} G_1 \Rightarrow_{\mathsf{f2f'}} G_2$ (Example 5.15, Figure 5.15 (p. 96)) is a counterexample.

Since the safety property is not even a 2-inductive invariant for a single graph rule, verifying it for the complete rule set of our running example – $\mathcal{R}' = \{\mathsf{f2f}', \mathsf{f2b}, \mathsf{b2s}, \mathsf{s2s}, \mathsf{s2a}', \mathsf{a2b}, \mathsf{a2f}'\}$ – will yield a similar result. It is no surprise that the safety property is not a 2-inductive invariant – or, in fact, an invariant for any possible value of k: The graph rules do not contain any mechanism to prevent acceleration and high speed modes when approaching a track. Even the non-trivial guaranteed constraint $\mathcal{H}$ of our running example (Example 5.1 (p. 69)) will not change this result. Although it will necessarily reduce the amount of counterexamples, some will remain, including the transformation sequence *trans*. △

Example 5.17 (2-inductive invariant checking for a safe system)**.** Now, consider a graph transformation system $GTS = (TG, \mathcal{R})$ with $\mathcal{R} = \{\mathsf{f2f}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{s2s}, \mathsf{s2a}, \mathsf{a2b}, \mathsf{a2f}\}$ (Example 5.1 (p. 69)) and (again) the safety property $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$. The rules f2f, a2f, and s2a are shown again in Figures 5.17(a), 5.17(b), and 5.17(c), respectively; F_1 is shown again in Figure 5.17(d). The rules shown have non-trivial application conditions in comparison to their (unsafe) counterparts a2f′, s2a′, and f2f′. Intuitively, they prevent acceleration or high speed modes – or rather, the application of the respective rules – when a switch is two tracks ahead (or, for s2a, one or two tracks ahead). We will also consider the non-trivial guaranteed constraint $\mathcal{H} = \bigwedge_{1 \le i \le 16} \neg H_i$ (Example 5.1 (p. 69)); H_1, H_5, and H_6 are depicted again in Figures 5.17(e), 5.17(f), and 5.17(g).

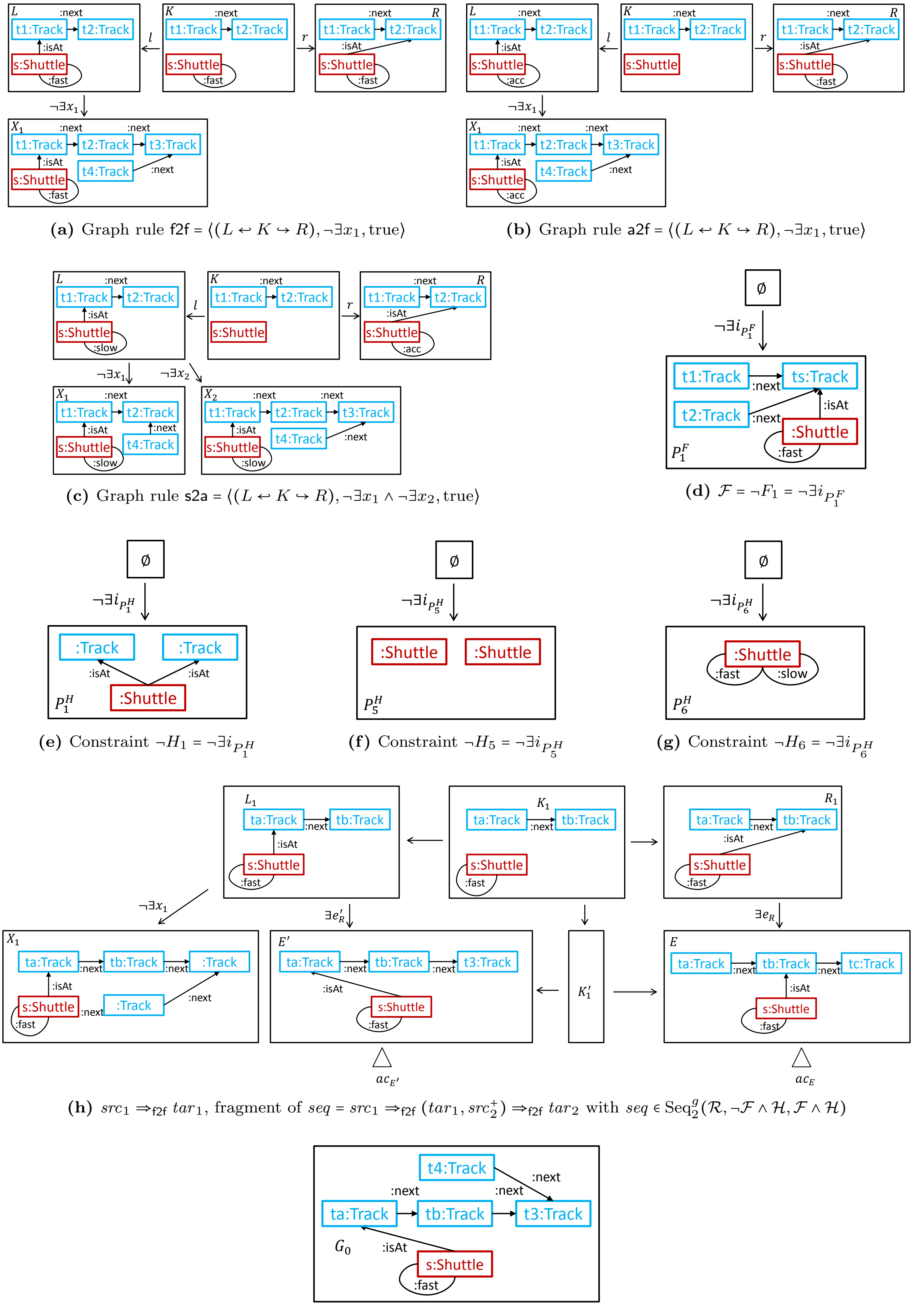
In order to apply Theorem T.2g (p. 96) and to determine whether $\mathcal{F}$ is a 2-inductive invariant of GTS under $\mathcal{H}$, we have to compute $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$. Due to the higher number of rules and constraints involved, the number and complexity of the resulting s/t-pattern sequences is significantly higher than in Examples 5.13 (p. 90) and 5.16 (p. 97). The main point, however, is that the negative application conditions of f2f, a2f, and s2a are taken into account in steps SC$_1$-3 and SC$_2$-3 of the Seq-construction. In particular, for all sequences

$$seq = src_1 \Rightarrow_{b_1} (tar_1, src_2^+) \Rightarrow_{b_2} tar_2 \quad \text{with} \quad seq \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$$

and $b_1, b_2 \in \mathcal{R}$, we know that the respective left application conditions ac_{L_i} have been added as a conjunctive operand to the source patterns src_1 and src_2 (with the latter being shifted src_2^+ as part of the target/source pattern). The left application conditions of s2s, a2b, f2b, and b2s are trivially true and do not have any effect on the construction. For s2a, f2f, and a2f, however, the left application conditions restrict the rules' applicability and hence, possible instantiations of the source patterns and s/t-pattern sequences.

Consider Figure 5.17(h), which depicts an s/t-pattern sequence $src_1 \Rightarrow_{\mathsf{f2f}} tar_1$. That sequence is a fragment of a 2-sequence of s/t-patterns $seq = src_1 \Rightarrow_{\mathsf{f2f}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$ with $seq \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$. This case is similar, if not equivalent, to the sequence discussed in Examples 5.13 (p. 90) and 5.15 (p. 95) and shown in Figure 5.14 (p. 92). There are two differences here: rule f2f has an additional negative application condition and there is a non-trivial guaranteed constraint $\mathcal{H}$. For clarity, the figure leaves out most nested conditions of the source and target pattern and only depicts them as $src_1 = \exists(e'_R, \dots) \wedge \neg\exists x_1$ and $tar_1 = \exists(e_R, \dots)$. Then, because $\neg\exists x_1$ contradicts existential conditions in $ac_{E'}$, the reduced source pattern $src_{1|\varnothing}$ cannot be satisfied by any graph.

In particular, consider graph G_0 from Example 5.15 (p. 95), depicted again in Figure 5.17(i). Recall that G_0 was a candidate to satisfy the leftmost source pattern's reduction to a constraint in the previous example and hence, the first graph of a transformation sequence that is a counterexample for $\mathcal{F}$ being a 2-inductive invariant. Here, we cannot find a match for the left side of the graph rule f2f such that it satisfies $\exists(e'_R, \dots) \wedge \neg\exists x_1$: the shuttle is not allowed to stay in speed mode fast when there is a switch two tracks ahead. Intuitively, this is also the reason for the absense of satisfying graphs for other leftmost source patterns of s/t-pattern sequences in $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$: we would have to find a situation where two subsequent



(a) Graph rule f2f = $\langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1, \text{true}\rangle$

(b) Graph rule a2f = $\langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1, \text{true}\rangle$

(c) Graph rule s2a = $\langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1 \wedge \neg\exists x_2, \text{true}\rangle$

(d) $\mathcal{F} = \neg\mathcal{F}_1 = \neg\exists i_{P_1^F}$

(e) Constraint $\neg H_1 = \neg\exists i_{P_1^H}$

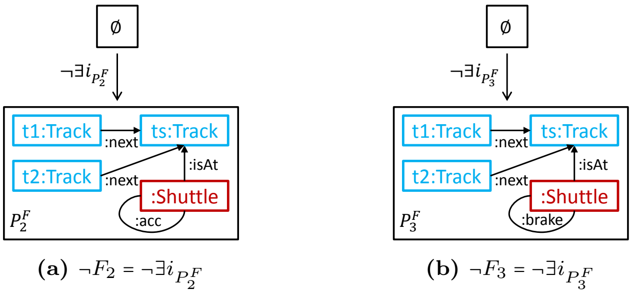
(f) Constraint $\neg H_5 = \neg\exists i_{P_5^H}$

(g) Constraint $\neg H_6 = \neg\exists i_{P_6^H}$

(h) $src_1 \Rightarrow_{\mathsf{f2f}} tar_1$, fragment of $seq = src_1 \Rightarrow_{\mathsf{f2f}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$ with $seq \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$

(i) Graph G_0 (cf. Example 5.15) with $G_0 \nvDash src_{1|\emptyset}$

Figure 5.17. – Fragments of an example system with a 2-inductive invariant $\mathcal{F} = \neg\mathcal{F}_1$



(a) $\neg F_2 = \neg\exists i_{P_2^F}$

(b) $\neg F_3 = \neg\exists i_{P_3^F}$

Figure 5.18. – Extended safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$

rule applications result in a (fast) shuttle on a switch; however, the first of those rules cannot be applied because of the negative application conditions in f2f, a2f, or s2a. In summary, $\mathcal{F}$ is a 2-inductive invariant for GTS under $\mathcal{H}$.

We can even extend our safety property to also forbid an accelerating or braking shuttle on a switch. The corresponding graph constraints $\neg F_2$ and $\neg F_3$ from Example 5.1 (p. 69) are shown again in Figure 5.18. Then, our new safety property is $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ – and $\mathcal{F}$ is a 2-inductive invariant for GTS under $\mathcal{H}$.

In this case, $\mathcal{F}$ being a 2-inductive invariant – as opposed to a larger value of k – is indirectly encoded in the rules' negative application conditions and the speed mode protocol. If the rules were to look ahead only one instead of two tracks and if speed modes could directly change to slow, the safety property might be a 1-inductive invariant. If we were to look ahead three or more tracks, we might have to check $\mathcal{F}$ as a 3-inductive invariant or use higher values for k. In practice, those 'numerical properties' of behavioral rules cannot usually be freely chosen. For example, having to look two tracks ahead instead of one might be rooted in a shuttle's braking distance: if a shuttle only registers switches that are only one track ahead, braking in time might not be possible. △

An important question is the choice of k for a given system. While it is possible to verify a graph constraint as a k-inductive invariant for iteratively increasing values of k, that approach only terminates when a k-inductive invariant is found or an upper bound for k is reached. Proper values for k, whether as start values or upper bounds, depend on the systems and application scenarios.

Theorem T.2g (p. 96) provide a means of verifying k-inductive invariants, which is the inductive step of Lemma L.1 (p. 65). Alone, a k-inductive invariant cannot be used to reason about the validity of the invariant in the state space of a graph grammar. Therefore, the following section will address the application of the Seq-construction to establish the base case of our inductive verification approach.

5.4. $k-1$-Bounded Backward Model Checking

In order to infer from a k-inductive invariant its validity for the state spaces of all graph grammars induced by a graph transformation system and a start configuration constraint, we need to establish the base case of the inductive argument described in Lemma L.1 (p. 65). This means showing the absence of a violation of $\mathcal{F}$ in the $k-1$-bounded state spaces of such graph grammars.

In particular, we recall the verifiable condition establishd by Lemma 5.3 (p. 75): we are looking for a transformation sequence $trans = G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_n$ such that $1 \leq n \leq k-1$, $G_0 \vDash \mathcal{S}$, $G_n \nvDash \mathcal{F}$, and $G_i \vDash \mathcal{H}$ with $1 \leq i \leq n$. In other words – and similar to the verification of k-inductive invariants – we are looking for a transformation sequence that leads to $\neg\mathcal{F}$

under $\mathcal{H}$. However, in contrast to Theorem T.2g (p. 96), we cannot only compute and consider $\mathrm{Seq}^g_{k-1}(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$. Rather, since any transformation sequence with a length between 1 and $k-1$ is a possible counterexample, we need to consider $\mathrm{SEQ}^g_k(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$, which contains all sequences in $\mathrm{Seq}^g_i(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$ for $1 \leq i \leq k-1$. Note that satisfaction of $\mathcal{S}$ by the first graph of a satisfying transformation sequence is not encoded as part of the Seq-construction, but will be included directly in the theorem.

Theorem T.3g (bounded backward model checking)**.** *Let $GTS = (TG, \mathcal{R})$ be a typed graph transformation system and $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \models \mathcal{F}$.*

For all graphs $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ and graph grammars $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$, we have $G \models \mathcal{F}$, if and only if for all sequences $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n) \in \mathrm{SEQ}^g_{k-1}(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$, there does not exist a graph G_0 with $G_0 \models src_{1|\varnothing} \wedge \mathcal{S}$.

Proof. We will show both directions separately.

If. We will show this direction by contraposition. Given an arbitrary graph grammar $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$, we assume the existence of a graph $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ such that $G \not\models \mathcal{F}$. By Lemma 5.3 (p. 75), there exists a transformation sequence $trans = G_0 \Rightarrow_{b_1, m_1, m'_1} \dots \Rightarrow_{b_n, m_n, m'_n} G_n$ (with $b_i \in \mathcal{R}$) that leads to $\neg\mathcal{F}$ (i.e. $G_n \not\models \mathcal{F}$) under $\mathcal{H}$ such that $G_0 \models \mathcal{S}$ and $1 \leq n \leq k-1$.

By Theorem T.1g (p. 85) (construction of s/t-pattern sequences), there is a s/t-pattern sequence $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n$ with $seq \in \mathrm{Seq}^g_n(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$ (and hence, $seq \in \mathrm{SEQ}^g_{k-1}(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$) such that $trans \models seq$. Then, $m_1 \models src_1$ and, by Lemma 2.38 (p. 43), we have $G_0 \models src_{1|\varnothing}$. Since $G_0 \models \mathcal{S}$ by assumption, we get $G_0 \models src_{1|\varnothing} \wedge \mathcal{S}$.

Only if. We will show this direction by contraposition. Assume the existence of an s/t-pattern sequence $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n) \in \mathrm{SEQ}^g_{k-1}(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$ such that there exists a graph G_0 with $G_0 \models src_{1|\varnothing} \wedge \mathcal{S}$. By Lemma 5.14 (p. 94), there is a transformation sequence $trans = G_0 \Rightarrow_{b_1, m_1, m'_1} \dots \Rightarrow_{b_n, m_n, m'_n} G_n$ such that $trans \models seq$. By Theorem T.1g (p. 85), $trans$ leads to $\neg\mathcal{F} \wedge \mathcal{H}$ under $\mathcal{H}$. Since, in particular, $G_0 \models \mathcal{S}$ and by Lemma 5.3 (p. 75), we have a graph G_n violating $\mathcal{F}$ with $G_n \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ and $GG \in \mathrm{IND}(GTS, \mathcal{S})$. □

The theorem describes an equivalence: false negatives and false positives cannot occur. However, similar to Theorem T.2g (p. 96), the underlying condition – whether there is a satisfying graph $G_0 \models src_{1|\varnothing} \wedge \mathcal{S}$ – is (in general) an undecidable problem.

While we have mainly introduced the process of bounded backward model checking as a means to verify the base case for an established k-inductive invariant, this is not the only application scenario. In general, bounded backward model checking allows us to find symbolic transformation sequences from a possible start graph to an erroneous state. Regardless of inductive invariant checking, these symbolic counterexamples or their concrete instantiations can be useful for analysis and debugging.

Example 5.18 (1-bounded backward model checking for an unsafe system)**.** Consider the constraint $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$ (Figure 5.17(d), p. 99) and rules $\mathcal{R} = \{\mathsf{s2s}, \mathsf{a2b}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{f2f}, \mathsf{s2f}, \mathsf{a2f}\}$ as before, and the guaranteed constraint $\mathcal{H} = \bigwedge_{1 \leq i \leq 16} \neg H_i$ (Examples 5.1 (p. 69) and 5.17 (p. 98)).

Furthermore, we choose a start configuration constraint $\mathcal{S} = \neg SC_1 = \neg\exists i_{P_1^{SC}}$ (Figure 5.19) that allows all start configurations that do not contain a shuttle driving in speed mode fast. Note that $\mathcal{S} \models \mathcal{F}$. The general idea behind this start configuration constraint is that system initialization with shuttles already driving fast is not plausible.

Given this system and start configuration constraint, we can find a single rule application from a possible start graph leading to a violation. Consider the s/t-pattern sequence $seq =$

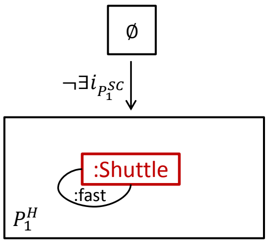


Figure 5.19. – Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$

$src_1 \Rightarrow_{\mathsf{a2f}} tar_1$ – with $seq \in \mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$ – which is depicted in Figure 5.20(a). In particular, the Seq-construction yields a target pattern

$$tar_1 = \bigvee_{i \in I} \exists t_1^i \wedge \mathrm{Shift}(i_{R_1}, \mathcal{H})$$

and a source pattern

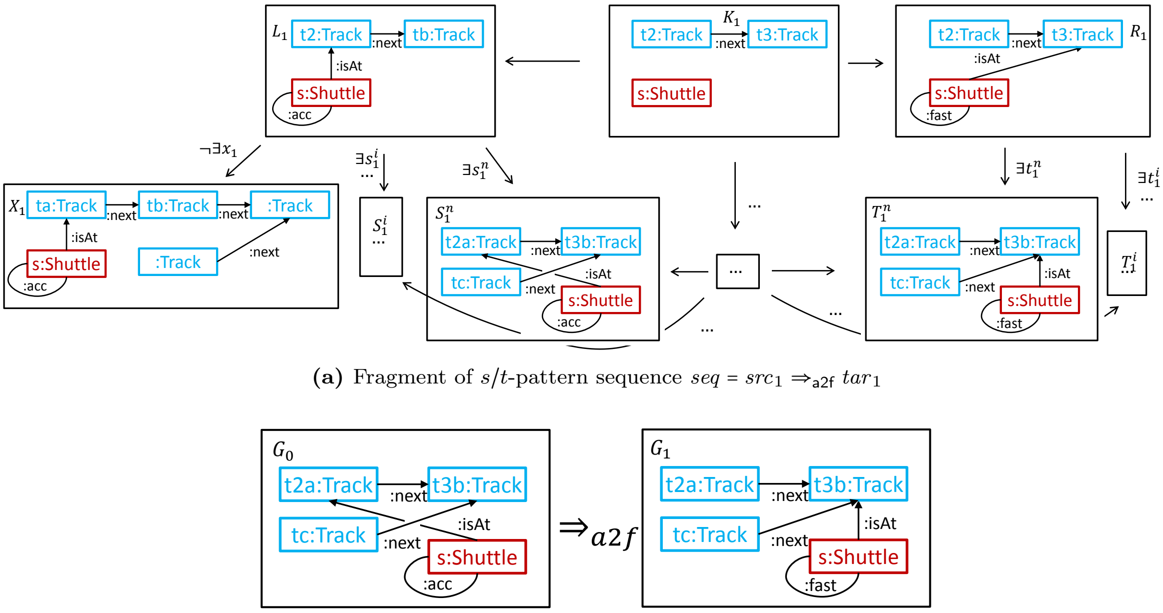
$$src_1 = \bigvee_{i \in I} \exists s_1^i \wedge \neg\exists x_1 \wedge \mathrm{L}(\mathsf{a2f}, \mathrm{Shift}(i_{R_1}, \mathcal{H})) \wedge \mathrm{Shift}(i_{L_1}, \mathcal{H}).$$

The disjunction in the target pattern is the result of shifting the safety violation, i.e. a (fast) shuttle located on a switch, to the right rule side; in the source pattern, it has been transferred over the rule a2f. The guaranteed constraint $\mathcal{H}$ has also been shifted to the right rule side – $\mathrm{Shift}(i_{R_1}, \mathcal{H})$ – because we specifically calculate $\mathrm{Seq}_1^g(\ldots, \neg\mathcal{F} \wedge \mathcal{H}, \ldots)$; however, it is not depicted in the figure. $\mathrm{Shift}(i_{R_1}, \mathcal{H})$ is then transferred to the source pattern over the left rule side by the L-construction. Since we require $\mathcal{H}$ to be fulfilled in all intermediate graphs in satisfying sequences ($\mathrm{Seq}_1^g(\ldots, \ldots, \mathcal{H})$), src_1 is extended by $\mathrm{Shift}(i_{L_1}, \mathcal{H})$ (not depicted). Finally, since a2f has a non-trivial application condition, $\neg\exists x_1$ is conjunctively joined to the source pattern.

The graph S_1^n and the corresponding morphism $s_1^n : L_1 \hookrightarrow S_1^n$ is part of the source pattern as an existential condition $\exists s_1^n$. As part of a disjunction, it shows one possibility for a situation leading to the violation after rule application, which is encoded as $\exists t_1^n$ in the target pattern. In particular, we have a shuttle located directly before a switch and driving in speed mode acc. Since the negative application condition only prevents acceleration if there is a switch two tracks ahead, the shuttle can drive forward, increase its speed mode to fast, and hence, cause a violation of the safety property.

Formally, by Theorem T.3g (p. 101), we are looking for a graph G_0 such that $G_0 \vDash src_{1|\varnothing} \wedge \mathcal{S}$, i.e. $G_0 \vDash src_{1|\varnothing} \wedge \neg SC_1$. Figure 5.20(b) depicts such a graph G_0 that isomorphic to S_1^n. In particular, we have $G_0 \vDash src_{1|\varnothing}$. Furthermore, since the shuttle in G_0 is not located on a switch, we also have $G_0 \vDash \neg SC_1$ and hence, $G_0 \vDash src_{1|\varnothing} \wedge \mathcal{S}$. By Theorem T.3g (p. 101), we know that there is a graph $G \in \mathrm{REACH}_1(GG, \mathcal{H})$ with $G \nvDash \mathcal{F}$ for a graph grammar $GG \in \mathrm{IND}(GTS, \mathcal{S})$. In fact, $GG = (GTS, G_0)$ is such a graph grammar: the transformation sequence $trans = G_0 \Rightarrow_{\mathsf{a2f}, m_1, m_1'} G_1$ (Figure 5.20(b)) leads to a violation of $\mathcal{F}$ under the guaranteed constraint $\mathcal{H}$. Since G_0 is a possible start configuration ($G_0 \vDash \mathcal{S}$), we have $G_1 \in \mathrm{REACH}_1(GG, \mathcal{H})$.

In summary, this is an example of a system where a violation of the safety property might occur directly after a rule application from a possible start graph. Although some graph grammars in the set of induced graph grammars might be safe, we cannot establish the validity of the safety property for all graphs in the 1-bounded state spaces of all induced graph grammars. In particular, we have constructed a symbolic encoding of possible error traces: transformation sequences leading to a violation within the bounded state space. Since we can find a violation for $\mathcal{F} = \neg F_1$, we will also find violations for the extended safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$,



(a) Fragment of s/t-pattern sequence $seq = src_1 \Rightarrow_{\mathsf{a2f}} tar_1$

(b) Transformation sequence $trans = G_0 \Rightarrow_{\mathsf{a2f'}, m_1, m_1'} G_1$

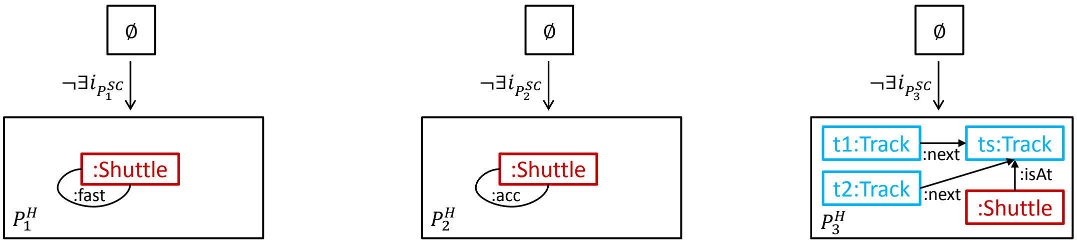
Figure 5.20. – Fragment of s/t-pattern sequence $seq \in \mathrm{Seq}_1^g(\mathcal{R}, \neg F \wedge H, \mathcal{H})$ and example transformation sequence $trans \vDash seq$

with F_2 and F_3 as in Example 5.1 (p. 69) and Example 5.17 (p. 98), Figures 5.18(a) and 5.18(b).

This is also an example for a system where, although a 2-inductive invariant has been verified (Example 5.17), the base case of the inductive argument could not be established. $\mathcal{F}$ is a 2-inductive invariant of GTS under $\mathcal{H}$: any transformation sequence of length 1 that is free of violations of the safety proptery and the guaranteeed constraint implies validity of the safety property in all possible subsequent states (provided they satisfy $\mathcal{H}$). However, as shown above, not all transformation sequences of length 1 from possible start graphs are free of violations. △

Example 5.19 (1-bounded backward model checking for a safe system)**.** Now, we extend the safety property with $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$; all rules and the guaranteed constraint remain unchanged. Figures 5.21(b) and 5.21(c) depict additional constraints $SC_2 = \exists i_{P_2^{SC}}$ and $SC_3 = \exists i_{P_3^{SC}}$; then, we choose the start configuration constraint as $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$. In the previous example, we found a symbolic error trace of length 1 from a start graph where a shuttle was driving in speed mode acc. With $\neg SC_2$, this situation cannot occur in valid start graphs. Indeed, although G_0 in Figure 5.20(b) still satisfies the (unchanged) reduced source pattern $src_{1|\varnothing}$, it does not satisfy $\neg SC_2$ because G_0 contains P_2^{SC} and hence, $G_0 \nvDash \neg SC_2$. Furthermore, there is no graph G_0 satisfying both the reduced leftmost source pattern and the (extended) start configuration constraint for any of the s/t-pattern sequences in $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$. Hence, we have $G \vDash \mathcal{F}$ for all graphs G in the bounded state spaces $\mathrm{REACH}_1(GG, \mathcal{H})$ for all graph grammars in $\mathrm{IND}(GTS, \mathcal{S})$.

Intuitively, this makes sense: since shuttle speed modes follow the protocol shown in Example 5.1 (p. 69), Figure 5.1(b), a shuttle in speed mode slow requires at least application of s2a and a2f to switch to speed mode fast – and more if it starts in speed mode brake. Hence, when possible start graphs only allow speed modes slow and brake, we cannot reach a graph with a fast shuttle after one transformation, let alone a fast shuttle on a switch. For an accelerating shuttle (F_2) on a switch, the application condition of s2a prevents reachability within one



(a) Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$ **(b)** Constraint $\neg SC_2 = \neg\exists i_{P_2^{SC}}$ **(c)** Constraint $\neg SC_3 = \neg\exists i_{P_3^{SC}}$

Figure 5.21. – Fragments of start configuration constraints

transformation step of a possible start graph. For a braking shuttle (F_3), the combination of all three start configuration constraint fragments prevents a violation.

Note that there are other ways of fixing the erroneous system from Example 5.18 (p. 101). For example, we could prohibit start graphs where shuttles are located on a switch or within one track of a switch. This modification of the start configuration constraint would also lead to the validity of $\mathcal{F}$ in the 1-bounded state spaces of all induced graph grammars. △

Although bounded backward model checking can be applied independently from k-inductive invariant checking, its main objective in this thesis is to provide the base case of our inductive verification approach. The combination of both steps will be discussed in the next section.

5.5. Operational Invariant Checking

Our verification approach is based on an inductive argument: in order to establish the validity of a graph constraint in all state spaces under a guaranteed constraint of all graph grammars induced by a graph transformation system and a start configuration constraint, the graph constraint has to be a k-inductive invariant and has to be valid in all $k{-}1$-bounded state spaces of all induced graph grammars. Symbolic verification of both preconditions is described in Theorem T.2g (p. 96) and Theorem T.3g (p. 101), respectively. Now, we combine both theorems as follows:

Theorem T.4g (operational invariant checking)**.** *Let $GTS = (\mathcal{R}, TG)$ be a typed graph transformation system and $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \vDash \mathcal{F}$.*

For all graph grammars $GG = (GTS, G_0)$ with $GG \in \text{IND}(GTS, \mathcal{S})$, $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ if there is a $k \geq 1$ such that the following conditions hold:

1. *For all sequences $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n) \in \text{SEQ}^g_{k-1}(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$, there does not exist a graph G with $G \vDash src_{1|\varnothing} \wedge \mathcal{S}$.*
2. *For all sequences $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k) \in \text{Seq}^g_k(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$, there does not exist a graph G such that $G \vDash src_{1|\varnothing}$.*

Proof. By precondition (1) and Theorem T.3g (p. 101), we have $G \vDash \mathcal{F}$ for all graphs $G \in \text{REACH}_{k-1}(GG, \mathcal{H})$ and all graph grammars $GG \in \text{IND}(GTS, \mathcal{S})$. By precondition (2) and Theorem T.2g (p. 96), $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ for GTS (2). Then, by Lemma L.1 (p. 65), all graphs $G \in \text{REACH}(GG, \mathcal{H})$ for all graph grammars $GG \in \text{IND}(GTS, \mathcal{S})$ satisfy $\mathcal{F}$, concluding the proof. □

Condition 1 – the base case of our inductive argument – follows Theorem T.3g (p. 101); condition 2, which is the inductive step, follows Theorem T.2g (p. 96). Formally, this theorem is just a different formalization of Lemma L.1 (p. 65). However, the inclusion of the Seq-constructions shows a constructive approach towards a possible implementation.

Contrary to both earlier theorems, the conjunction of their respective conditions is a sufficient condition only – this theorem does not describe an equivalence. In particular, we cannot conlcude from the failure to establish a k-inductive invariant its violation in a specific graph in a graph grammar's state space.

Example 5.20 (operational invariant checking for an unsafe system)**.** Examples for systems where the safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ cannot be established as an operational invariant via Theorem T.4g (p. 104) are Example 5.16 (p. 97) and Example 5.18 (p. 101).

In the former example, $\mathcal{F}$ is not a 2-inductive invariant or, although not shown there, an invariant for any value of k. However, without further analysis, we cannot be sure $\mathcal{F}$ is not an operational invariant; i.e. that there exists a graph in a graph grammar's state space violating $\mathcal{F}$. As mentioned before, Theorem T.4g does not provide a necessary condition to verify operational invariants.

In the latter example, there is a transformation sequence (Figure 5.20(b), p. 103) leading to a violation of $\mathcal{F}$ after a rule application to a possible start graph. Hence, $\mathcal{F}$ is not an operational invariant for the system described in that example. △

Example 5.21 (operational invariant checking for a safe system)**.** In Examples 5.17 (p. 98) and 5.19 (p. 103), we have used a system with the rule set $\mathcal{R} = \{\mathsf{f2f}, \mathsf{a2f}, \mathsf{s2a}, \mathsf{s2s}, \mathsf{f2b}, \mathsf{a2b}, \mathsf{b2s}\}$, the safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$, a guaranteed constraint $\mathcal{H} = \bigwedge_{1 \le i \le 15} \neg H_i$, and a start configuration constraint $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$. We have shown via Theorems T.2g (p. 96) and T.3g (p. 101) that $\mathcal{F}$ is a 2-inductive invariant for $GTS = (TG, \mathcal{R})$ under $\mathcal{H}$ and that $\mathcal{F}$ is valid in all 1-bounded state spaces of graph grammars in $\mathrm{IND}(GTS, \mathcal{S})$. In combination and by Theorem T.4g, $\mathcal{F}$ is satisfied in all graphs of all state spaces of the induced graph grammars under the constraint $\mathcal{H}$. △

Theorem T.4g is a constructive approach to implementing Lemma L.1 (p. 65), which requires a k-inductive invariant under a guaranteed constraint $\mathcal{H}$ and the invariant's validity in $k-1$-bounded state spaces under $\mathcal{H}$. However, validity of $\mathcal{H}$ is not explicitly proven. If validity is not assumed by the problem description – for example, because it is an external assumption not part of the problem domain – explicit verification may be required. Lemma L.2 offered a technique to do that:

Lemma L.2 (validity of constraints in induced graph grammars under a constraint [4])**.** *Let $GG = (GTS, G_0)$ be a graph grammar with a graph transformation system $GTS = (TG, \mathcal{R})$ and let $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \vDash \mathcal{F}$ and $\mathcal{S} \vDash \mathcal{H}$. $\mathcal{F}$ is an operational invariant of* $\mathrm{REACH}(GG)$ *for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$, if the following conditions hold:*

0. $\mathcal{H}$ is a 1-inductive invariant for GTS.
1/2. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$.

Under Lemma L.2, the state spaces under $\mathcal{H}$ are equivalent to the state spaces without that restriction. This approach uses 1-inductive invariant checking to establish the validity of $\mathcal{H}$. As explained before, the idea is that the guaranteed constraint is easier to prove as an invariant; in particular, $\mathcal{H}$ is supposed to be a 1-inductive invariant and, as a precondition, it should be implied by the start configuration constraint $\mathcal{S}$. Similar to $\mathcal{S}$ and $\mathcal{F}$, this can be achieved by choosing a desired start configuration constraint $\mathcal{S}'$ and defining the final start configuration constraint as $\mathcal{S} = \mathcal{S}' \wedge \mathcal{H}$.

Then, the following theorem describes a constructive approach to Lemma L.2:

Theorem T.5g. *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \models \mathcal{F}$ and $\mathcal{S} \models \mathcal{H}$.*

For all graph grammars $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$, $\mathcal{F}$ is an operational invariant of GG if the following conditions hold:

0. *For all sequences $seq = (src_1 \Rightarrow_{b_1} tar_1) \in \mathrm{Seq}^g_1(\mathcal{R}, \neg\mathcal{H}, \mathcal{H})$, there does not exist a graph G such that $G \models src_{1|\varnothing}$.*
1. *For all sequences $seq = (src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k) \in \mathrm{SEQ}^g_{k-1}(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{H})$, there does not exist a graph G with $G \models src_{1|\varnothing} \wedge \mathcal{S}$.*
2. *For all sequences $seq = (src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k) \in \mathrm{Seq}^g_k(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$, there does not exist a graph G such that $G \models src_{1|\varnothing}$.*

Proof. By Theorem T.2g (p. 96) – and appropriate subsitutions – $\mathcal{H}$ is a 1-inductive invariant for GTS. Also by Theorem T.2g (p. 96), $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$. By Theorem T.3g (p. 101), $\mathcal{F}$ is valid in all $k-1$-bounded state spaces of graph grammars in $\mathrm{IND}(GTS, \mathcal{S})$ under $\mathcal{H}$. By Lemma L.2 (p. 66), we get $G \models \mathcal{F}$ for all graphs G with $G \in \mathrm{REACH}(GG)$ and $GG \in \mathrm{IND}(GTS, \mathcal{S})$. □

Condition (0) establishes $\mathcal{H}$ as an operational invariant for all graph grammars induced by the graph transformation system and the start configuration constraint $\mathcal{S}$. Since $\mathcal{S}$ implies $\mathcal{H}$, all possible start configurations as the base case of the inductive argument will satisfy the guaranteed constraint. If $\mathcal{H}$ is a 1-inductive invariant – the inductive step of the argument – $\mathcal{H}$ will hence be valid in all graphs of all state spaces.

Similar to Theorem T.4g, conditions (1) and (2) are required to verify $\mathcal{F}$ as an operational invariant under $\mathcal{H}$. However, since $\mathcal{H}$ is an operational invariant for the unrestricted state spaces, $\mathrm{REACH}(GG)$ and $\mathrm{REACH}(GG, \mathcal{H})$ are identical per induced graph grammar $GG \in \mathrm{IND}(GTS, \mathcal{S})$.

It may seem counterintuitive to have both Theorem T.4g (p. 104) and Theorem T.5g. However, there are situations where verification of the guaranteed constraint is not possible and, indeed, not necessary. As explained in Chapter 4, we consider three types of guaranteed constraints: type graph constraints, external assumptions, and controlled execution. While it is desirable to have a type graph constraint verified as an operational invariant, it is also possible to prevent its violation by controlled execution, such as with an interpreter controlling the application of graph rules. In the latter case, direct verification is not possible and, as long as we trust the controlloing unit, not necessary.

Likewise, external assumptions often cannot be verified because they are, by definition, not part of the system. Consider a system that has a single fault assumption – not because two or more faults cannot happen, but because the probability of their occurence is under a certain threshold. System analysis disregards events (s/t-pattern sequences) with more than one fault, but their absence – i.e. their validity as an operational invariant – cannot be proven. Hence, depending on the system and type of guaranteed constraint, both Theorem T.4g (p. 104) and Theorem T.5g have their application scenarios.

Example 5.22 (operational invariant checking and guaranteed constraint)**.** Consider Example 5.21 (p. 105) with an extended start configuration constraint $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3 \wedge \mathcal{H}$. Then, $\mathcal{S} \models \mathcal{H}$ as required by Theorem T.5g. The results with respect to conditions (1) and (2) of Theorem T.4g (p. 104) and condition (1/2) of Lemma L.2 (p. 66) still hold: $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ and $\mathcal{F}$ is valid in every graph in $\mathrm{REACH}_{k-1}(GG, \mathcal{H})$ for graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$ – for both the former and extended start configuration constraint. Furthermore, although not shown here, $\mathcal{H}$ is a 1-inductive invariant of GTS. Then, by Theorem T.5g, $\mathcal{F}$ is an operational invariant of GG for all induced graph grammars

$GG \in \mathrm{IND}(GTS, \mathcal{S})$. This example is one case where verification of $\mathcal{H}$ is, indeed, possible: $\mathcal{H}$ specifies only type graph constraints that are preserved by the graph rules and can be verified. Here, no external assumptions or controlled execution play into the choice of $\mathcal{H}$. △

5.6. Discussion and Conclusion

This concludes the general approach to inductive invariant checking and the formulaic implementation of our verification approach motivated and explained in Chapter 4. The examples have illustrated three limitations and issues of our approach (which overlap with each other):

Computational effort. As Example 5.13 has demonstrated, the s/t-pattern sequences and their individual source patterns, target patterns, and target/source patterns quickly become unmanageable for humans even for systems with small to average numbers of constraints and graph rules. This is detrimental to both performance (of an implementation) and clarity of results, i.e. counterexamples. While s/t-pattern sequences can be instantiated (by Lemma 5.14 (p. 94)), this process is computationally challenging as well. Furthermore, having a symbolic representation of counterexamples for a k-inductive invariant or a symbolic error trace from potential start graphs to an error state is useful. Then, the result could be inspected by a human actor in order to adjust system behavior such that the verification succeeds. Besides the exponential growth of potential rule applications with increasing value of k, a main problem is the transfer of the graph constraint $\mathcal{H}$ or even $\mathcal{F} \wedge \mathcal{H}$ to all intermediate patterns in the construction. In particular, this may lead to redundancy in the patterns, which results in unnecessary complexity that further accumulates with increasing values of k. Algorithms to eliminate redundancy are costly to execute; it is usually preferable to avoid redundancy in the first place.

Satisfiability. Lemma 5.14 (p. 94) provides a means of finding satisfying transformation sequences to an s/t-pattern sequence – however, it relies on a similar procedure that finds a satisfying graph to a graph constraint. We have not provided a means of implementing the latter in algorithmic terms, i.e. have not provided a construction that produces a satisfying graph. There is existing work [Pen08a, SLO17] concerning that process, but the problem is both undecidable in general and, depending on the complexity of the graph constraint, computationally challenging. Still, we require satisfiability in Theorems T.2g-T.5g. For example, in Theorem T.2g (p. 96), unsatisfiable s/t-pattern sequences resulting from a Seq-construction are false negatives, indicating that the supposed violation of a k-inductive invariant cannot occur in an actual transformation sequence.

Termination. By construction, the Shift-construction and L-construction always provide finite results; since the Seq-construction only uses those two constructions (and logical conjunctions), it yields finite results for fixed values of k as well. However, finding a satisfying graph to a graph constraint (see above) is an undecidable problem; hence, the same holds for finding a satisfying transformation sequence to an s/t-pattern sequence. In that situation, an implementation has to make a decision between risking non-termination and incompleteness.

All three issues will be addressed in the next chapter, where we will discuss a restricted formal model [Dyc12] intended to alleviate the effects of the problems above.

Although the properties **Appl.-soundness**, **Appl.-termination**, etc. are primarily meant for the restricted approach and its implementation (Chapter 6), we will also apply them here for the sake of comparison. Note that **Appl.-deg.completeness** and **Appl.-performance** are

Table 5.2. – Properties of the general approach from a formal perspective

	Property	**Appl.-soundness**	**Appl.-termination**	**Appl.-deg.-completeness**	**Appl.-performance**
General approach	k-inductive invariant checking	(✓)	×	?/(✓)	?
	$k-1$-bounded backward model checking	(✓)	×	?/(✓)	?
	Operational invariant checking	(✓)	×	?/×	?

Table 5.3. – Properties of the general approach for a hypothetical implementation

	Property	**Appl.-soundness**	**Appl.-termination**	**Appl.-deg.-completeness**	**Appl.-performance**
General approach	k-inductive invariant checking	✓	(✓)	?/×	?
	$k-1$-bounded backward model checking	✓	(✓)	?/×	?
	Operational invariant checking	✓	(✓)	?/×	?

not intended to be binary properties: the former is concerned with the degree of completeness – the number of false negatives – not whether the approach is or is not complete. This degree cannot be determined for the general approach without an implementation and evaluation. Thus, we will denote it with a question mark in the respective tables, with an answer to the related binary question after the slash. Performance cannot be considered here; however, as dicussed above, computational effort is expected to be unreasonably high.

Judging soundness, termination, and completeness is difficult because of the implications of the underlying undecidable problem. We will consider two perspectives here: the formalization of the approach and a hypothetical implementation.

The former (formal) perspective is shown in Table 5.2. Arguably, when considering its formalization, the general approach is sound and complete with respect to k-inductive invariant checking and $k-1$-bounded backward model checking: Theorems T.2g (p. 96), T.3g (p. 101), and related theorems and lemmas describe necessary and sufficient criteria to find exactly the set of valid counterexamples. However, those criteria are not decidable in general and hence, the procedure may not terminate. For operational invariant checking, the approach is also incomplete (cf. Theorem T.4g (p. 104)).

In practice and implementation, failure to terminate may cause the approach to be unsound and incomplete in general: without a result, systems cannot be classified as either safe or unsafe. Hence, Table 5.3 shows the perspective of a hypothetical implementation: by enforcing termination (e.g. with a threshold on runtime) and classifying s/t-pattern sequences without a definitive answer as counterexamples and the respective system as unsafe, the general approach is sound, terminates, and is incomplete. This applies to k-inductive invariant checking, $k-1$-bounded backward model checking, and operational invariant checking.

Having the option to enforce termination and ensure soundness in exchange for completeness is partly a result of the counterexample-driven nature of the general approach. The construction of s/t-pattern sequences (Theorem T.1g (p. 85)) always terminates, yields a finite result, and is sound in the sense that it provides all possible counterexamples. Only their analysis (by finding satisfying transformation sequences, Lemma 5.14 (p. 94)) results in an undecidable problem. Hence, returning all s/t-pattern sequences built by the Seq-construction as counterexamples always ensures soundness and termination.

For systems where the respective satisfiability problems turn out to be decidable, both k-inductive invariant checking and $k{-}1$-bounded backward model checking are sound, complete, and terminate with respect to both their formalization and a hypothetical implementation. Operational invariant checking, however, remains incomplete: Theorem T.4g (p. 104) contains a sufficient condition, not a necessary one.

6. Restricted Approach to k-Inductive Invariant Checking

In Chapter 5, we have seen that even for a system with a small number of elements, visual and computational complexity of the Seq-construction and subsequent analysis quickly becomes unmanageable. However, specification of example systems does not necessarily require support for application conditions and graph constraints of arbitrary nesting and complexity. In this chapter, we will reintroduce a restricted formal model established and extended in earlier work [BBG$^+$06, Dyc12]. It has been shown to be sufficient to model systems with reasonable depth *[1, 3, 6]*. Following the introduction of the restricted formal model, we will discuss the formalizations and algorithms required for **Formal-restricted** and **Impl.-restricted**.

Before introducing the restricted formal model, we will reiterate the general formal model used in the general approach (Chapter 5):

Formal Model (general)**.** *Systems and system specifications consist of the following elements:*

System metamodels *are specified by type graphs (Definition 2.8 (p. 21)).*

System states *– including initial states – are described by typed graphs (Definitions 2.1 (p. 20) and 2.8 (p. 21)).*

System behavior *is described by a typed graph transformation system (Definition 2.24 (p. 34)), which consists of typed graph transformation rules (Definition 2.20 (p. 32)).*

Properties *are modeled as graph constraints (Definition 2.12 (p. 25)) – this includes the constraint $\mathcal{F}$ to be verified, the guaranted constraint $\mathcal{H}$, and the start configuration constraint $\mathcal{S}$. As an additional requirement, $\mathcal{S}$ needs to imply $\mathcal{F}$, i.e. $\mathcal{S} \models \mathcal{F}$.*

Systems *are specified by typed graph grammars (Definition 2.28 (p. 36)), which consist of an initial state – a start graph – and a typed graph transformation system.*

System state spaces *are described by the state spaces (Definitions 2.28 (p. 36) and 4.1 (p. 55)) – the set of all reachable graphs – of the corresponding graph grammars (under the guaranteed constraint).*

System sets *are described by graph grammars induced (Definition 4.7 (p. 59)) by a graph transformation system and a start configuration constraint.*

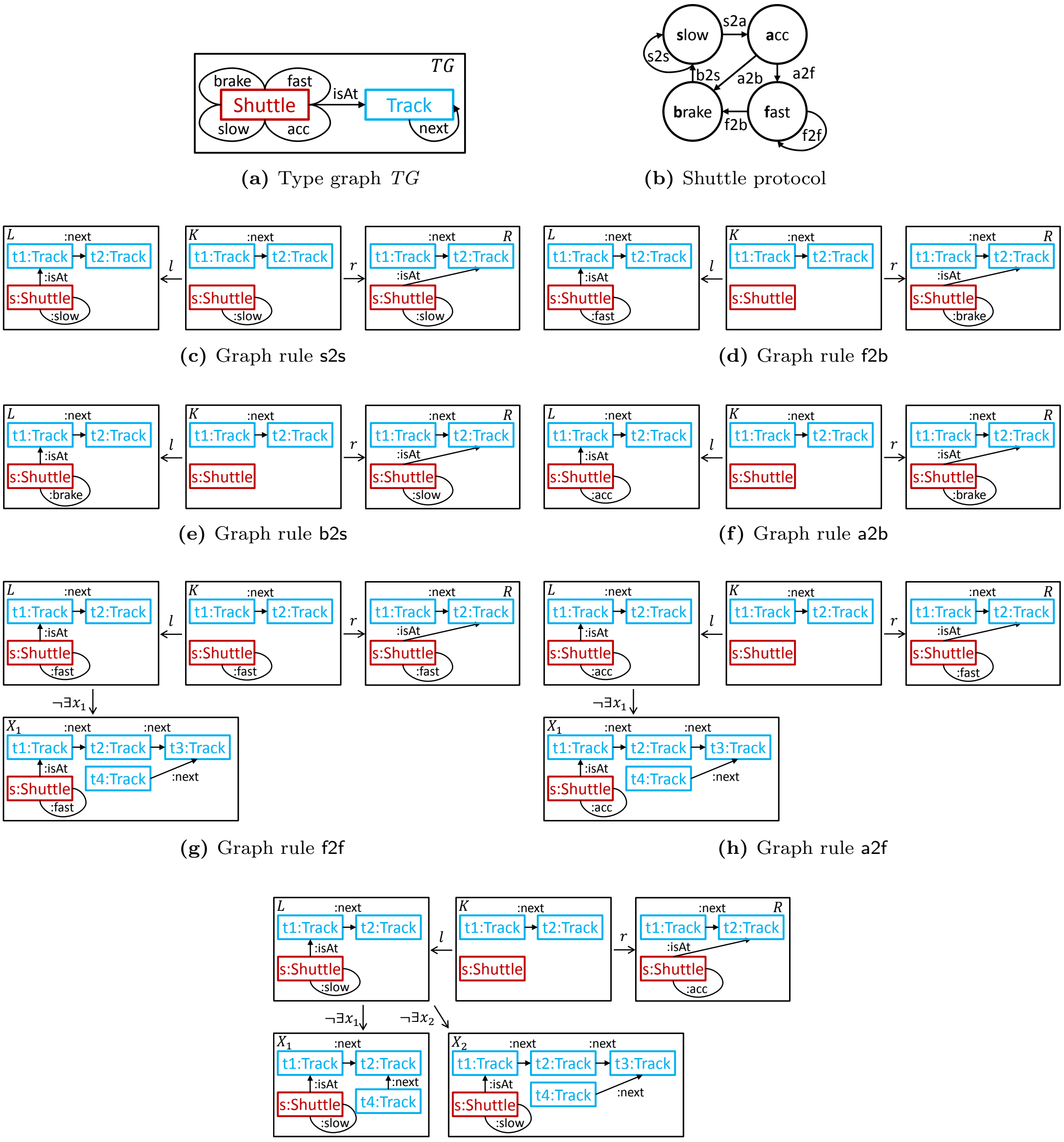
Similarly, we recall the central verification question from Chapter 4:

Verification Problem VP.1g. *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and graph constraints $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{H}$ with $\mathcal{S} \models \mathcal{F}$, does every graph grammar $GG \in \mathrm{IND}(GTS, \mathcal{S})$ have $\mathcal{F}$ as an operational invariant under $\mathcal{H}$?*

Example 6.1 (running example)**.** We will reuse the running example from Chapter 5, Example 5.1 (p. 69). Figure 6.1 relists the type graph TG (Figure 6.1(a)), the speed mode transition protocol (Figure 6.1(b)), and the set of graph rules $\mathcal{R} = \{\mathsf{s2s}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{a2b}, \mathsf{f2f}, \mathsf{a2f}, \mathsf{s2a}\}$ (Figures 6.1(c)-6.1(i)).

Again, we also occasionally use three alternative rules for f2f, a2f, and s2a – f2f′, a2f′, and s2a′ (Figure 6.2). These rules will again lead to an unsafe system. We will denote the associated graph transformation system and set of graph rules as $GTS' = (TG, \mathcal{R}')$ and $R' = \{\mathsf{s2s}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{a2b}, \mathsf{f2f'}, \mathsf{a2f'}, \mathsf{s2a'}\}$.

Figure 6.3 shows the graph constraint $\mathcal{F}$ to verify, a guaranteed constraint $\mathcal{H}$, and three fragments of a start configuration constraint. We want to verify the permanent absence of shuttles driving on a switch in speed modes fast, acc, or brake (Figures 6.3(a)–6.3(c)), i.e.



(a) Type graph TG

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

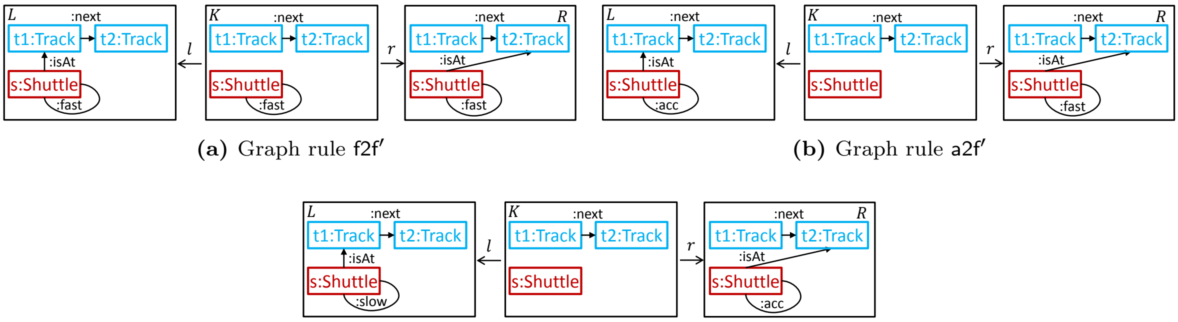
(f) Graph rule a2b

(g) Graph rule f2f

(h) Graph rule a2f

(i) Graph rule s2a

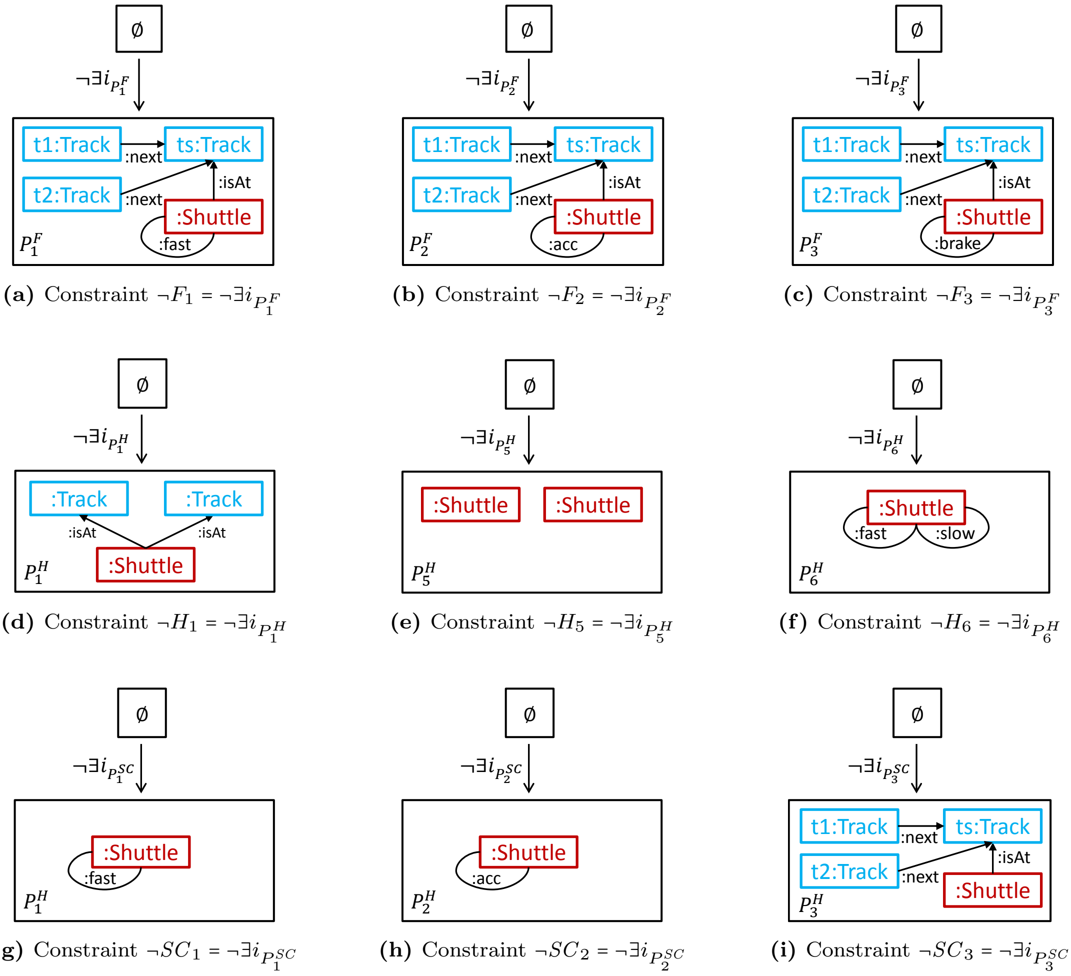
Figure 6.1. – Graph transformation system $GTS = (TG, \mathcal{R})$



(a) Graph rule f2f′ **(b)** Graph rule a2f′

(c) Graph rule s2a′

Figure 6.2. – Alternative rules f2f′, a2f′, and s2a′ without application conditions



(a) Constraint $\neg F_1 = \neg\exists i_{P_1^F}$ **(b)** Constraint $\neg F_2 = \neg\exists i_{P_2^F}$ **(c)** Constraint $\neg F_3 = \neg\exists i_{P_3^F}$

(d) Constraint $\neg H_1 = \neg\exists i_{P_1^H}$ **(e)** Constraint $\neg H_5 = \neg\exists i_{P_5^H}$ **(f)** Constraint $\neg H_6 = \neg\exists i_{P_6^H}$

(g) Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$ **(h)** Constraint $\neg SC_2 = \neg\exists i_{P_2^{SC}}$ **(i)** Constraint $\neg SC_3 = \neg\exists i_{P_3^{SC}}$

Figure 6.3. – Safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$, fragments of guaranteed constraint $\mathcal{H} = \neg\mathcal{H}_1 \wedge \ldots \wedge \neg H_{15}$, and parts of start configuration constraint $\mathcal{S}$

$\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$. Figures 6.3(d)-6.3(f) reiterate three fragments of the system's guaranteed constraint $\mathcal{H} = \neg H_1 \wedge ... \wedge \neg H_{15}$. Finally, Figures 6.3(g), 6.3(h), and 6.3(i) show fragments of the start configuration constraint $\mathcal{S}$. Depending on whether or not we limit verification to system states (and state spaces) under the guaranteed constraint $\mathcal{H}$, we can choose $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$ or $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3 \wedge \mathcal{H}$. In the latter case, we have $\mathcal{S} \vDash \mathcal{H}$ by construction. We can easily see that $\neg SC_1 \vDash \neg F_1$ and $\neg SC_2 \vDash \neg \mathcal{F}_2$; then, $\mathcal{S} \vDash \mathcal{F}$ as required. If it were not obvious – which, in general, is to be expected – we could join $\mathcal{F}$ conjunctively to $\mathcal{S}$ to form a new start configuration constraint.

All elements of the example are also listed in Sections C.1.2 and C.1.1 of Appendix C.

With respect to Verification Problem VP.1g (p. 60), we wonder whether all graphs in the state spaces of all graph grammars induced by the graph transformation system $GTS = (TG, \mathcal{R})$ (or $GTS' = (TG, \mathcal{R}')$) and the start configuration constraint $\mathcal{S}$ fulfill the safety property. In other words, starting from a state satisfying our start configuration constraint and following the behavior specified by the graph rules, can the system reach a state where a shuttle reaches a switch while driving in mode fast, accelerating, or braking? △

As before, we will use k-inductive invariant checking and its proof obligations formalized in Chapter 4:

Lemma L.1 (validity of constraints in graph grammars). *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and let $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \vDash \mathcal{F}$. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$, if there exists a $k \in \mathbb{N}$ with $k \geq 1$ such that the following conditions hold:*

1. $\forall GG(GG \in \mathrm{IND}(GTS, \mathcal{S}) \Rightarrow \forall G(G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H}) \Rightarrow G \vDash \mathcal{F}))$.
2. *$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.*

Again, we will focus on verifying state spaces under a guaranteed constraint. Similar to the general approach, we will introduce an approach that combines the results for restricted state spaces with verification of $\mathcal{H}$ at the end of this chapter.

Outline. This chapter focuses on a restricted verification approach following a restricted formal model that was established in earlier work and is reiterated in Section 6.1. In short, graph rules will only be allowed to have so-called composed negative application conditions: a conjunctive combination of negated existential conditions without further nesting. Likewise, we will use *composed graph patterns* to describe a safety property, guaranteed constraint, and start configuration constraint. A composed graph pattern is a conjunctive combination of negated graph patterns, which consist of an existential condition with a nested composed negative application condition.

Based on this restricted formal model, this chapter refines definitions, theorems, and algorithmic constructions established in Chapter 5. As such, it focuses on contribution **Formal-restricted** – the formalization of our restricted approach to verification with k-inductive invariants. Furthermore, as a part of **Impl.-restricted**, we present an idea of the approach's implementation in pseudocode. Figure 6.4 provides a rough overview of this chapter's sections and the formal elements introduced.

In Section 6.2, we will first rearrange the condition for the inductive step of Lemma L.1 as the new Lemma 6.10. Likewise, Lemma 6.11 will rearrange the condition for the base case such that the condition is more intuitive from an algorithmic perspective. In particular, both lemmas will provide conditions that can be falsified by finding an appropriate transformation sequence as a counterexample. Also, both lemmas are special cases of the corresponding Lemmas 5.2 and 5.3 from the general approach, respectively.

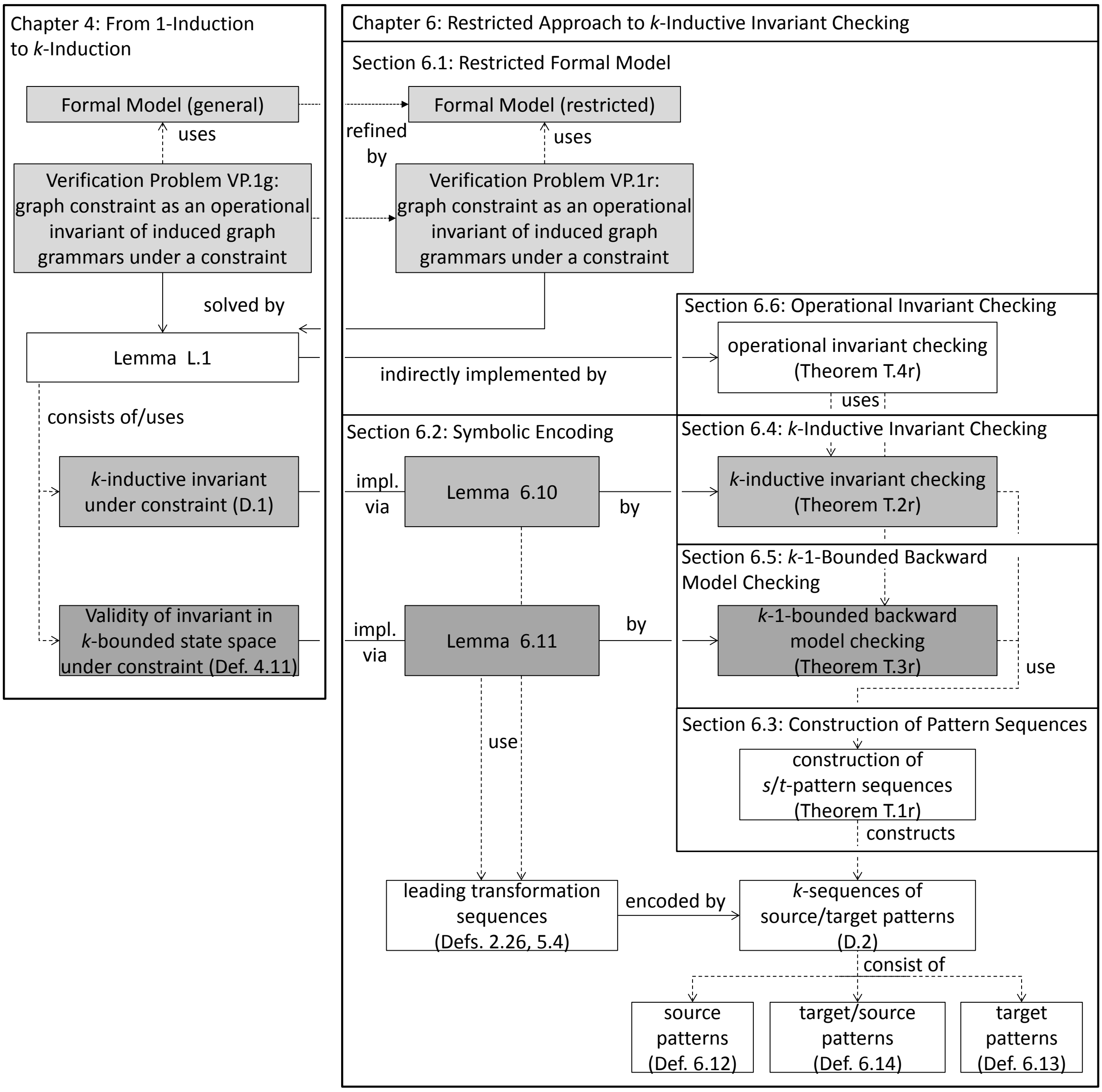

Chapter 4: From 1-Induction to *k*-Induction
Formal Model (general)
uses
Verification Problem VP.1g: graph constraint as an operational invariant of induced graph grammars under a constraint
solved by
Lemma L.1
consists of/uses
k-inductive invariant under constraint (D.1)
Validity of invariant in *k*-bounded state space under constraint (Def. 4.11)
Chapter 6: Restricted Approach to *k*-Inductive Invariant Checking
Section 6.1: Restricted Formal Model
refined by
Formal Model (restricted)
uses
Verification Problem VP.1r: graph constraint as an operational invariant of induced graph grammars under a constraint
indirectly implemented by
Section 6.6: Operational Invariant Checking
operational invariant checking (Theorem T.4r)
uses
Section 6.2: Symbolic Encoding
impl. via
Lemma 6.10
by
Section 6.4: *k*-Inductive Invariant Checking
k-inductive invariant checking (Theorem T.2r)
Section 6.5: *k*-1-Bounded Backward Model Checking
impl. via
Lemma 6.11
by
k-1-bounded backward model checking (Theorem T.3r)
use
Section 6.3: Construction of Pattern Sequences
construction of *s*/*t*-pattern sequences (Theorem T.1r)
constructs
use
leading transformation sequences (Defs. 2.26, 5.4)
encoded by
k-sequences of source/target patterns (D.2)
consist of
source patterns (Def. 6.12)
target/source patterns (Def. 6.14)
target patterns (Def. 6.13)

Figure 6.4. – Overview and dependencies of definitions, theorems, and lemmas

Furthermore, the elements of the symbolic encoding established in Section 5.1 will be refined in Section 6.2. Again, transformation sequences will be encoded in *s/t-pattern sequences* (Definition D.2), which can represent an infinite number of transformation sequences. S/t-pattern sequences (of length k) for the restricted formal model still consist of a *source pattern*, a *target pattern*, and between 0 and $k-1$ *target/source patterns*. However, given the restrictions for application conditions and graph constraints, source patterns, target patterns, and target/source patterns can be restricted to a special form of application conditions (Definitions 6.12, 6.13, and 6.14). The basic idea remains unchanged: a source pattern encodes a match of a rule and its potential application, a target pattern encodes a comatch, and a target/source pattern encodes the combination of both. An s/t-pattern sequence then combines elements of the three types and can be satisfied by transformation sequences.

Section 6.3 will refine the construction of s/t-pattern sequences introduced in Section 5.2. In particular, Theorem T.1r defines a specialized version of the Seq-construction, which takes a set of graph rules and a graph pattern as parameters. This construction is tailored to create s/t-pattern sequences (of a specified length) whose satisfying transformation sequences use rules from the given set of graph rules and lead to the graph pattern passed to the construction. In contrast to the general Seq-construction, we do not supply a second graph constraint to be fulfilled by intermediate graphs of a satisfying transformation sequence. We will still use the refined Seq-construction to create s/t-pattern sequences that (symbolically) represent all transformation sequences that may serve as counterexamples for the conditions in Lemma 6.10 and Lemma 6.11. However, since satisfiability of constraints by intermediate graphs is not taken into account by the construction, this aspect needs to be considered in the analysis of the s/t-pattern sequences (counterexamples).

In Section 6.4, Theorem T.2r then applies the Seq-construction (Theorem T.1r) to reason about the validity of a k-inductive invariant by verifying the condition established by Lemma 6.10. Likewise, Section 6.5 introduces Theorem T.3r, which reasons about the base case – the proposed invariant's validity in the induced graph grammars' k-bounded state space. Both Theorems create a number of s/t-pattern sequences as counterexamples and then attempt to discard them using the conditions in Lemma 6.10 or Lemma 6.11, respectively.

Note the difference to the general approach: in the Seq-construction for the restricted formal model, we only use the information of which graph pattern the represented transformation sequences should lead to. Satisfiability of graph constraint – the composed guaranteed pattern, for example – by intermediate graphs is analyzed in a restricted fashion in Theorems T.2r and T.3r. Furthermore, we do not attempt to find a satisfying transformation sequence for s/t-pattern sequences (cf. Lemma 5.14). Both steps reduce computational effort and improve performance. Even more important, they allow us to guarantee termination in our implementation. However, that comes at a cost: since the problem is, in general, still undecidable and since our approach also guarantees soundness, it may be incomplete – s/t-pattern sequences as counterexamples may be *false negatives*, i.e. may erroneously classify a system as unsafe.

Section 6.6 then combines k-inductive invariant checking and the invariant's validity in the k-bounded state space, i.e. the inductive step and base case of our inductive verification approach. Theorem T.4r does not bring any new considerations, but uses the conditions and constructions of Theorem T.2r and Theorem T.3r. It establishes formal justification for our approach to verifying the conditions of Lemma L.1.

In Section 6.7, we will sketch the implementation of the formal constructions and theorems in the restricted approach from an algorithmic perspective.

Finally, Section 6.8 discusses the results of this chapter and raises open issues.

6.1. Restricted Formal Model

In Chapter 5, we have encountered the problem of exponential complexity in typical computations involving graphs and application conditions. In particular, the Seq-construction of the general approach quickly becomes unamanagable in terms of both readability and computational effort. The driving factor of complexity is the Shift-construction: intuitively, we find all overlappings between two graphs, which requires finding all possible common subgraphs. Since the number of subgraphs rises exponentially with the number of nodes alone, so does the complexity of the Shift-construction.

This situations worsens if we allow nesting of application conditions to an arbitrary depth – which is possible by the definition of nested application conditions. However, this degree of arbitrary complexity does not necessarily bring a benefit in many application scenarios. Where behavior is specified (as graph rules) by human designers, we are unlikely to encounter nested conditions exceeding a certain depth.

For these reasons, previous work on inductive invariang checking for graph transformation systems has introduced a restricted formal model [BBG$^+$06, BG08b], which has been gradually extended [Dyc12]. The goal of imposing restrictions with respect to our formal model of system specifications is to strike a balance between expressive specifications and computational effort for the algorithms involved in our verification process. While a high degree of freedom in specifications allows the applicability of our approach in a larger number of scenarios, the resulting computational effort is detrimental to performance (of an implementation). As a result, the approach may be impractical to verify to certain scenarios even if requirements on expressive power are met. A more specialized and restricted structure of graph rules, graph constraints, and application conditions will later allow us to optimize our implementation with respect to those restrictions. With respect to the content and contribution of this thesis, the restricted formal model is the basis of **Formal-restricted** and **Impl.-restricted** – the formal approach and justification, and later implementation, of k-inductive invariant checking. Therefore, in the following, we will reiterate said restricted formal model [Dyc12] and take at look at the motivation behind the restrictions.

We can identify several important characterstics of system elements in our running example:

Example 6.2 (form of graph rules and graph constraints in running example)**.** Four rules of our running example conform to the basic form of $\mathsf{s2s} = \langle (L \hookleftarrow K \hookrightarrow R), \mathrm{true}, \mathrm{true}\rangle$ or $\mathsf{f2b} = \langle (L \hookleftarrow K \hookrightarrow R), \mathrm{true}, \mathrm{true}\rangle$ with a trivial left application condition true. Even where we use non-trivial application conditions – such as in $\mathsf{s2a} = \langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1 \wedge \neg\exists x_2, \mathrm{true}\rangle$ (Figure 6.1(i)) – the condition consists of only two negated existential conditions without nested subconditions.

A similar observation can be made for graph constraints. Until now, we have only discussed constraints consisting of a negated existential condition consisting of only one morphism such as in Figures 6.3(a) and 6.3(d). More intuitively, our graph constraints have only described forbidden subgraphs.

However, thusly limiting graph constraints may be too restrictive. We might want to have a safety property with a conditional character: when certain safeguard mechanisms are in place, a property that would otherwise constitute a safety violation could be considered safe. For example, a shuttle driving fast on a track might be allowed if it is equipped with an additional safety feature. Such a construct would require a negated existential condition nested within the first condition: a certain subgraph may not exist unless another subgraph (or a combination of subgraphs) exists. △

The restricted formal model [Dyc12] is based on these ideas in more general terms:

1. Application conditions in rules must have the trivial form true or consist of conjunctively joined negated existential conditions without nested subconditions.
2. Graph constraints must consist of only one negated existential condition with a nested application condition similar to those allowed in rules.

This idea of restricting application conditions has been formalized as a *composed negative application condition*:

Definition 6.3 (composed negative application condition [Dyc12])**.** *A* composed negative application condition *over a graph A is an application condition over A of the form $ac = \text{false}$, $ac = \text{true}$, or $ac = \bigwedge_{i\in I} \neg\exists a_i$ for morphisms $a_i : A \hookrightarrow A_i$. An individual application condition $\neg\exists a_i$ is called a* negative application condition*.*

One or more negated existential conditions may be conjunctively joined, but are not allowed to have further nested application conditions. In comparison to unrestricted nested application conditions (Definition 2.10 (p. 22)), this is a limitation, although not as restrictive as the formal model of earlier work on inductive invariant checking [BBG⁺06].

For the general notion of graph rules introduced in Definition 2.20 (p. 32) – and in our general approach – graph rules may have (left) nested application conditions of arbitrary depth and arbitrary logical combinations. Here, rules may only have a (left) composed negative application condition. Note that composed negative application conditions cannot contain disjunctions of conditions or non-negated existential conditions (positive application conditions).

Also, as explained above, it makes sense to use safety properties that consist of a single negated existential condition with a second (nested) composed negative application condition. Then, a violation can be described as an existential condition with a nested composed negative application condition. This structure is called a *graph pattern*, or simply *pattern*:

Definition 6.4 (graph pattern [Dyc12])**.** *A* graph pattern*, or* pattern*, is a graph constraint of the form $C = \text{true}$, $C = \text{false}$, or $C = \exists(i_A, ac)$ with i_A an injective morphism $i_A : \varnothing \hookrightarrow A$ and ac a composed negative application condition over A.*

Note that the term graph pattern may be confused with source pattern or target pattern (Definitions 5.7 (p. 77) and 5.8 (p. 77)), which are part of the symbolic encoding for both the general and restricted formal model. While source and target patterns are application conditions over a rule side, graph patterns are graph constraints. Sometimes, if the context is clear, we will just refer to graph patterns as patterns.

If a graph pattern's application condition *ac* is trivially true, it simply describes the existence of the graph that is the morphism's codomain. We will sometimes refer to this graph as the pattern's (positive) context. Otherwise, we require conditional existence: the graph should be present unless it can be extended to one of the codomains in the composed negative application condition.

As before, graph patterns and composed negative application conditions are required to be in injective normal form: morphisms in subconditions must be injective. Since the initial morphism $i_A : \varnothing \hookrightarrow A$ from the empty graph $\varnothing$ to an arbitrary graph A will always be injective, this is not an issue for graph patterns without nested application conditions. For composed negative application conditions in graph rules and patterns, on the other hand, this is, indeed, a restriction. However, given graph rules with injective matching, we will only need to consider satisfiability of application conditions by injective morphisms. Since injective morphisms are closed under composition and decomposition, a limitation to conditions in injective normal form does not affect expressive power in our case: subconditions with non-injective morphisms cannot be satisfied by injective morphisms; thus, they will be equivalent to false for the purpose of satisfiability by injective morphisms.

Finally – where application conditions and graph constraints are concerned – we specify how graph patterns appear as properties to be verified for the systems under verification. In particular, we combine negated graph patterns in conjunctions as composed forbidden patterns:

Definition 6.5 (composed graph pattern [Dyc12])**.** *A* composed graph pattern*, or* composed pattern*, is a conjunctive combination of negated graph patterns, i.e. a graph constraint of the form $\mathcal{C} = \bigwedge_{i \in I} \neg C_i$ for graph patterns C_i.*

If we use a composed graph pattern as a safety property to be verified, we say it is a composed forbidden (graph) pattern *and say that its individual graph patterns are* forbidden (graph) patterns*.*

If we use a composed graph pattern as a guaranteed constraint, we say it is a composed guaranteed (graph) pattern *and say that its individual graph patterns are* guaranteed (graph) patterns*.*

If we use a composed graph pattern as a start configuration cosntraint, we say it is a composed start configuration (graph) pattern *and say that its individual graph patterns are* start configuration (graph) patterns*.*

Note that patterns are negated (and conjunctively joined) in a composed pattern. While the term pattern alone usually refers to the non-negated existential condition, the terms forbidden pattern or guaranteed pattern usually imply that a pattern appears in negated form. As such, a forbidden pattern forbids the occurrence of the non-negated pattern; a guaranteed pattern guarantees or assumes the absence of the non-negated pattern.

The restriction of application conditions and graph constraints to composed negative application conditions and graph patterns will allow us to implement certain optimizations in our verification approach. However, their application requires the preservation of the structures in the restricted formal model under certain transformations. In earlier work [Dyc12], these properties have been shown for the Shift- and L-constructions.

Fact 6.6 (Shift-construction and composed negative application conditions [Dyc12])**.** *Given a composed negative application condition $ac_N = \bigwedge_{i \in I} \neg\exists(x_i : N \hookrightarrow X_i)$ with graphs N, X_i, injective morphisms x_i and an index set I, for each graph N' and each injective morphism $n : N \hookrightarrow N'$, the application condition $ac'_N = \mathit{Shift}(n, ac_N)$ is a composed negative application condition.*

Fact 6.7 (L-construction and composed negative application conditions [Dyc12])**.** *Given a composed negative application condition $ac = \bigwedge_{i \in I} \neg\exists(x_i : R \hookrightarrow X_i)$, for each graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R) ac_L, \mathit{true} \rangle$, $L(b, ac)$ is a composed negative application condition.*

Our restricted verification approach will frequently require comparisons of graph patterns with respect to implication (Definition 2.36 (p. 42)). However, as explained in Chapter 2, the definition does not supply a constructive approach. Due to the inherent undecidability of the problem, algorithms for the general case are either unsound, incomplete, or may not terminate.

Nevertheless, taking into account our restricted formal model, we can make a reasonable effort to checking implication of graph patterns. In Example 2.37 (p. 42) in Chapter 2, we have reduced the question of implication of graph constraints – graph patterns, in fact – to subgraph relationships. For patterns without a nested composed negative application condition, the existence of that relationship is equivalent to a pattern implying another pattern; then, that problem is decidable [Pen09]. However, since we allow our patterns to be equipped with composed negative application conditions, we have to consider a more general case. The following theorem established this general and constructive perspective for the verification of graph pattern implication in earlier work:

Theorem 6.8 (implication of patterns [Dyc12])**.** *Let $C = \exists(i_P : \varnothing \hookrightarrow P, ac)$ and $C' = \exists(i_{P'} : \varnothing \hookrightarrow P', ac')$ be two patterns, with composed negative application conditions $ac = \bigwedge_{i \in I} \neg\exists(x_i : P \hookrightarrow X_i)$ and $ac' = \bigwedge_{j \in J} \neg\exists(x'_j : P' \hookrightarrow X'_j)$ for index sets I, J. Then $C' \vDash C$, if the following condition is fulfilled:*

1'. There exists a $j \in J$ such that x'_j is bijective or

1. There exists an injective morphism $m : P \hookrightarrow P'$ such that:

2. With Shift$(m, \neg\exists x_i) = \bigwedge_{k \in K_i} \neg\exists(x''_{ik} : P' \hookrightarrow X''_{ik})$ for a number of corresponding index sets K_i, for each x_i it holds that $\forall k(k \in K_i \Rightarrow \exists j \exists y(y : X'_j \hookrightarrow X''_{ik} \wedge x''_{ik} = y \circ x'_j))$.

$$\begin{tikzcd}
P \arrow[rr, hook, "m"'] \arrow[d, hook, "x_i"] & & P' \arrow[dl, hook, "x''_{ik}"] \arrow[d, hook, "x'_j"'] \\
X_i \arrow[r, hook, "m'_{ik}"'] & X''_{ik} & X'_j \arrow[l, hook', "y"] \arrow[ul, phantom, "=" near start]
\end{tikzcd}$$

Proof. Assuming condition (1') holds, consider an arbitrary graph G' with $g' : P' \hookrightarrow G'$ and thus, $G' \vDash \exists i_{P'}$. Since x'_j (for the specific j) is bijective, there is a $q' : X'_j \hookrightarrow G'$ with $q' \circ x'_j = g'$. Hence, we have $g' \nvDash \exists x'_j$ and, more importantly, $G' \nvDash C'$. Thus, there does not exist a graph G' with $G' \vDash C'$ and consequently, $\forall G(G \vDash C' \Rightarrow G \vDash C)$ is trivially true.

Assuming condition (1') does not hold, (1) and (2) hold by precondition; for the proof, we refer to the respective source [Dyc12]. □

Note that the theorem describes an implication, not an equivalence: not being able to meet the conditions does not guarantee that C' does not imply C. This will play into the implementation as well: not attempting to get a definite answer to the question of implication will be the cost of having a sound and terminating algorithm. Since the problem is undecidable [Pen09], the algorithm cannot be complete. Again, the problem is decidable (both in theory and by the above theorem) if the implied pattern has the trivial composed application conditions true. More generally, the problem of implication of constraints is decidable for the non-nested fragment of constraints – i.e. boolean combinations of constraints $\exists(a, \text{true})$ [Pen09].

The implementation of Theorem 6.8 (used in previous work [Dyc12]) is shown in Algorithm 6.1. Intuitively, the comparison is still mainly based on subgraph relationships. First, we make sure that the (potentially) implying pattern is satisfiable (condition (1'), lines 1-3). If one of its negative application conditions $\neg\exists x'_j$ (which make up the composed negative application condition) is described by an isomorphism x'_j, C' cannot be satisfied: any satisfying graph G' that contains P' will then also contain the isomorphic X'_j. Then, because the graph pattern is equivalent to false and cannot be satisfied by any graph, by definition, it implies all other graph patterns (and even all unrestricted graph constraints).

Second, the implying pattern's existential condition's graph (P' in $\exists i'_P$) should have the implied pattern's existential condition's graph (P in $\exists i_P$) as a subgraph (condition (1), line 4). This is related to the argument explained in Example 2.37 (p. 42). If there is no such subgraph relationship, we have the case that the graph P' satisfies $C' = \exists(i'_P, ac')$, but does not satisfy $C = \exists(P, ac)$ (if it did, P' would need to contain P as a subgraph). Hence, C' does not imply C.

Lastly, condition(2), lines 5-13 take care of the pattern's composed negative application conditions. Even with a subgraph relationship described by the injective morphism $m : P \hookrightarrow P'$, it is still possible for C' to not imply C because of the patterns' composed negative application conditions. Specifically, there might exist a graph G that contains P' – and hence P – as a subgraph such that P' cannot be extended to violate any of the negative application conditions $\neg\exists x'_j$ but that P can be extended to violate at least one of the negative application conditions $\neg\exists x_i$. Then, G satisfies C', but not C and hence, C' does not imply C.

Thus, assuming satisfaction of C' and hence, of the composed negative application condition ac', (2) attempts to verify satisfaction of the composed negative application condition ac. In order to compare ac and ac', the individual negative application conditions in ac – $\neg\exists x_i$ – are transferred via $m : P \hookrightarrow P'$ to the context of ac', which is the implying pattern's existential condition's graph P'. This is achieved by computation of Shift$(m, \neg\exists x_i)$. If, for all

Algorithm 6.1: implies(C', C)

```
desc.  : implementation of Theorem 6.8
input  : a pattern $C = \exists(i_P, ac_P)$ and a pattern $C' = \exists(i'_P, ac'_P)$
output: whether $C'$ was found to imply $C$ according to Theorem 6.8

foreach condition $\neg\exists x'$ in $ac'_P$ do
    if $x'$ is bijective then                                   /* case (1') */
        return true
foreach injective morphism $m : P \hookrightarrow P'$ do            /* case (1/2) */
    matchedAllConditions ← true
    foreach condition $\neg\exists x''$ in Shift($m, ac_P$) do
        matchedCondition ← false
        foreach condition $\neg\exists x'$ in $ac_{P'}$ do
            if there is a $y : X'_j \hookrightarrow X''$ such that $x'' = y \circ x'$ then
                matchedCondition ← true
                break
        if not matchedCondition then
            macthedAllConditions ← false
            break
    if macthedAllConditions then
        return true
return false
```

of the transferred negative application conditions $\neg\exists x''_{ik}$ (the result of the Shift-operation), ac' contains a stronger or equally strong negative application condition, C' implies C. Such a stronger or equally strong condition can again be expressed by using subgraph relationships (cf. $y : X'_j \hookrightarrow X''_{ik}$). Intuitively and informally, a negative application condition is stronger in the sense that it excludes more graphs if it has fewer elements (nodes and edges); then, a graph containing another graph as a subgraph is less strict than its subgraph when used in a negative application condition.

Note that there may be several injective morphisms $m : P \hookrightarrow P'$. Hence, conditions (1) and (2) must be checked for all those morphisms. If one m can be found such that condition (2) holds, C' implies C. Since there are finite numbers of potential morphisms $m : P \hookrightarrow P'$ and negative application conditions in ac and ac' and since the Shift-construction always yields finite results, Theorem 6.8 can be implemented as an algorithm that always terminates. However, the computation may still be costly, particularly because of the Shift-construction and subgraph computation involved. Furthermore, since pattern implication in general is undecidable, the algorithm, which is sound, cannot be complete – and hence, the theorem does not describe an equivalence: even if no suitable morphism m can be found, C' may still imply C.

An example for the application of Theorem 6.8 to two patterns is given in Example B.1 in Appendix B.

This concludes the explanation of concepts required by our restricted formal model, which is introduced as follows:

Formal Model (restricted)**.** *Systems and system specifications consist of the following elements:*

System metamodels *are specified by type graphs.*

Table 6.1. – Comparison of general and restricted formal model

Element/ model	Meta-models	System states	Systems	System sets	System state spaces
General formal model	Type graphs	Typed graphs	Typed graph grammars	Induced (typed) graph grammars	Graph grammars' state spaces under guaranteed constraint
Restricted formal model	Type graphs	Typed graphs	Typed graph grammars	Induced (typed) graph grammars	Graph grammars' state spaces under guaranteed constraint

Element/ model	System behavior	Safety properties	Guaranteed properties	Initial states
General formal model	Graph rules with left nested application conditions	Nested graph constraint(s)	Nested graph constraint(s)	Nested graph constraint(s)
Restricted formal model	Graph rules with left composed negative application conditions	Composed graph pattern	Composed graph pattern	Composed graph pattern

System states – *including initial states – are described by typed graphs.*

System behavior *is described by a typed graph transformation system, which consists of typed graph transformation rules. Rules may only have a left application condition in the form of a composed negative application condition (Definition 6.3 (p. 118)).*

Properties *are modeled as composed graph patterns (Definitions 6.5 (p. 119) and 6.4). More specifically, we specify a composed forbidden pattern $\mathcal{F}$ to be verified under a composed guaranteed pattern $\mathcal{H}$; in addition, start graphs are described by a composed start configuration pattern $\mathcal{S}$. Similar to the general approach, we require $\mathcal{S} \models \mathcal{F}$.*

Systems *are specified by typed graph grammars, which consist of an initial state – a start graph – and a typed graph transformation system.*

System state spaces *are described by the state spaces – the set of all reachable graphs – of the corresponding graph grammars (under the composed guaranteed pattern).*

System sets *are described by graph grammars induced by a graph transformation system and a composed start configuration pattern.*

The differences between the general and restricted formal model are shown in Table 6.1.

Example 6.9 (running example, composed negative application conditions, graph patterns)**.** Our running example conforms to the restricted formal model. In particular,

- rules s2s, f2b, b2s, a2b (Figures 6.1(c)-6.1(f), p. 112) have the trivial composed negative application condition true,
- rules f2f, a2f, s2a (Figures 6.1(g)-6.1(i), p. 112) have (non-trivial) composed negative application conditions of the form $\neg\exists x_1$ or $\neg\exists x_1 \wedge \neg\exists x_2$, respectively,
- the alternative rules f2f, a2f, s2a (Figures 6.2(a)-6.2(c), p. 113) have the trivial composed negative application condition true,
- the safety property is a composed forbidden pattern $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ (Figures 6.3(a)-6.3(c), p. 113),

- the guaranteed constraint is a composed guaranteed pattern $\mathcal{H} = \bigwedge_{1 \leq i \leq 16} \neg H_i$ (with fragments shown in Figures 6.3(d)-6.3(f), p. 113,
- and the start configuration constraint is a composed start configuration pattern $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$ (or $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3 \wedge \mathcal{H}$) (Figures 6.3(g)-6.3(i), p. 113). △

Using this restricted formal model requires modifying Verification Problem VP.1g as follows:

Verification Problem VP.1r. *Given a graph transformation system $GTS = (\mathcal{R}, TG)$ with rules of the form $b_i = \langle (L_i \hookleftarrow K_i \hookrightarrow R_i), ac_{L_i}, \text{true}\rangle$ for composed negative application conditions ac_{L_i} and composed graph patterns $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{H}$ with $\mathcal{S} \vDash \mathcal{F}$, does every graph grammar $GG \in \text{IND}(GTS, \mathcal{S})$ have $\mathcal{F}$ as an operational invariant under $\mathcal{H}$?*

In the following section, we will discuss how the symbolic encoding for the general approach can be refined to take the characteristics of the restricted formal model into account.

6.2. Symbolic Encoding

As mentioned earlier, we aim to apply Lemma L.1 (p. 65), which relies on an inductive argument with k-inductive invariants as the inductive step and their validitiy in $k{-}1$-bounded state spaces of the respective induced graph grammars as the base case. We recall the definition of k-indcutive invariants:

Definition D.1 (k-inductive invariant [3]). *Given a typed graph transformation system $GTS = (\mathcal{R}, TG)$ and graph constraints $\mathcal{F}$ and $\mathcal{H}$, $\mathcal{F}$ is a k-*inductive invariant *of GTS* under *$\mathcal{H}$, if, for all sequences of transformations to $\mathcal{R}$ $trans = G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_k$ it holds that:*

$$(\forall z (0 \leq z \leq k \Rightarrow G_z \vDash \mathcal{H}) \wedge \forall z (0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})) \quad \Rightarrow \quad (G_k \vDash \mathcal{F})$$

However, given our the restriction of safety properties and the guaranteed constraint to composed graph patterns, we know that $\mathcal{F}$ and $\mathcal{H}$ are composed graph patterns and follow a specific structure. In particular, $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ and $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$ for graph patterns F_i and H_i. This allows us to introduce a variant of Lemma 5.2 (p. 75) – similar to the general approach, we can establish a more constructive form of our proof obligation for the verification of k-inductive invariants.

Lemma 6.10 (k-inductive invariant and transformation sequences as counterexamples [3]). *Given a typed graph transformation system $GTS = (TG, \mathcal{R})$, a composed forbidden pattern $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$, and a composed guaranteed pattern $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$, $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$, if and only if the following holds for each k-sequence of transformations to $\mathcal{R}$ $trans = G_0 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_k$:*

$$\exists u (G_k \vDash F_u) \Rightarrow (\exists z, v (0 \leq z \leq k \wedge G_z \vDash H_v) \vee \exists z, v (0 \leq z \leq k-1 \wedge G_z \vDash F_v))$$

Proof. We can rearrange the formula from Definition D.1 (for all sequences of transformations to $\mathcal{R}$ $trans = G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_k$):

$$\begin{aligned} & (\forall z (0 \leq z \leq k \Rightarrow G_z \vDash \mathcal{H}) \wedge \forall z (0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})) \Rightarrow (G_k \vDash \mathcal{F}) \\ \Longleftrightarrow & (G_k \nvDash \mathcal{F}) \Rightarrow \neg(\forall z (0 \leq z \leq k \Rightarrow G_z \vDash \mathcal{H}) \wedge \forall z (0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F})) \\ \Longleftrightarrow & (G_k \nvDash \mathcal{F}) \Rightarrow (\exists z (0 \leq z \leq k \wedge G_z \nvDash \mathcal{H}) \vee \exists z (0 \leq z \leq k-1 \wedge G_z \nvDash \mathcal{F})) \\ \Longleftrightarrow & \exists u (G_k \vDash F_u) \Rightarrow (\exists z, v (0 \leq z \leq k \wedge G_z \vDash H_v) \vee \exists z, v (0 \leq z \leq k-1 \wedge G_z \vDash F_v)) \end{aligned}$$

□

The idea is to reduce complexity by splitting up a composed forbidden pattern into its individual forbidden patterns. Since those patterns are conjunctively combined, a potential violation occurs when a transformation sequence leads to one of the patterns F_i, which is described by the implication's precondition ($\exists u(G_k \vDash F_u)$). Then, to discard that transformation sequence as a counterexample, violations of an individual forbidden pattern earlier in the sequence ($\exists z, v(0 \leq z \leq k-1 \wedge G_z \vDash F_v)$) or violations of an individual guaranteed pattern ($\exists z, v(0 \leq z \leq k \wedge G_z \vDash H_v)$) have to be found.

Note that this lemma is different from its counterpart (Lemma 5.2 (p. 75)) when seen from an algorithmic perspective. Here, we look for transformation sequences as potential counterexamples (i.e. leading to a pattern F_n), which may then be discarded after further analysis (finding violations elsewhere in the sequence). In contrast to that, Lemma 5.2 (p. 75) states that certain sequences should not exist, without rearranging it into the implicative condition used in Lemma 6.10.

The structure already hints at the algorithm we will use in our verification procedure. Here, we will construct s/t-pattern sequences leading to $\neg\mathcal{F}$ – or rather, the individual patterns F_i. Then, in contrast to the general approach, the s/t-pattern sequences will be analyzed for occurences of forbidden or guaranteed patterns. In our general approach (Sections 5.2 and 5.3), those checks were integrated into the construction of the s/t-pattern sequences by transferring the respective constraints to the source and target patterns. The final analysis (Theorem T.2g (p. 96)) was only concerned with finding satisfying transformation sequences. Here, we aim to reduce computational effort by not integrating the other forbidden patterns and guaranteed patterns into the s/t-pattern sequences. Instead, their effects are considered in a a less costly analysis step.

Besides that, the desired effect of splitting the composed forbidden pattern is twofold: first, if we construct s/t-pattern sequences whose satisfying transformation sequences lead to an individual forbidden pattern as opposed to a composed forbidden pattern, the resulting sequence and its computation will be less complex; instead, however, there will be more s/t-pattern sequences. This plays into the second point: multiple s/t-sequences could, in theory, be subject to analysis in parallel.

Lemma 6.10 refines Lemma 5.2 (p. 75); both reason about k-inductive invariants. Similarly, we refine Lemma 5.3 (p. 75), which reasoned about $k-1$-bounded state spaces and transformation sequences as counterexamples.

Lemma 6.11 (k-bounded state spaces and transformation sequences as counterexamples)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o\in O} \neg SC_o$ be composed graph patterns with $\mathcal{S} \vDash \mathcal{F}$. For all graphs $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ with graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$, we have $G \vDash \mathcal{F}$ if and only if the following holds for each sequence of transformations to $\mathcal{R}$ trans = $G_0 \Rightarrow_{b_1,m_1,m'_1} \ldots \Rightarrow_{b_n,m_n,m'_n} G_n$ with $0 \leq n \leq k-1$:*

$$\exists u(G_n \vDash F_u) \Rightarrow (\exists z, v(0 \leq z \leq n \wedge G_z \vDash H_v) \vee \exists v(G_0 \vDash SC_v))$$

Proof. Given a graph grammar $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$, by definition of k-bounded state spaces (Definition 4.11 (p. 63)), the existence of a graph $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ with $G \nvDash \mathcal{F}$ is equivalent to the existence of a transformation sequence to $\mathcal{R}$ *trans* = $G_0 \Rightarrow_{b_1,m_1,m'_1} \ldots \Rightarrow_{b_n,m_n,m'_n} G_n$ with $1 \leq n \leq k-1$ and $G_n \nvDash \mathcal{F}$ where $G_0 \vDash \mathcal{S}$ and where all traversed graphs satisfy $\mathcal{H}$, i.e. $\forall i(1 \leq i \leq n \Rightarrow G_i \vDash \mathcal{H})$. $G_n \nvDash \mathcal{F}$ is equivalent to $\exists u(G_n \vDash F_u)$. Then, the absence of such a sequence with $\exists u(G_n \vDash F_u) \wedge \forall i(1 \leq i \leq n \Rightarrow G_i \vDash \mathcal{H}) \wedge G_0 \vDash \mathcal{S}$ is equivalent to validity of $\mathcal{F}$ in all $k-1$-bounded state spaces of the graph grammars induced by GTS and $\mathcal{S}$ – and we can express the absence by requiring for all transformation sequences to $\mathcal{R}$ *trans* = $G_0 \Rightarrow_{b_1,m_1,m'_1} \ldots \Rightarrow_{b_n,m_n,m'_n} G_n$:

$$\begin{aligned}
&\neg(\exists u(G_n \vDash F_u) \wedge \forall i(1 \le i \le n \Rightarrow G_i \vDash \mathcal{H}) \wedge G_0 \vDash \mathcal{S})\\
&\iff \neg(\exists u(G_n \vDash F_u)) \vee \neg(\forall i(1 \le i \le n \Rightarrow G_i \vDash \mathcal{H}) \wedge G_0 \vDash \mathcal{S})\\
&\iff \exists u(G_n \vDash F_u) \Rightarrow \neg(\forall i(1 \le i \le n \Rightarrow G_i \vDash \mathcal{H}) \wedge G_0 \vDash \mathcal{S})\\
&\iff \exists u(G_n \vDash F_u) \Rightarrow (\exists z, v(0 \le z \le n \wedge G_z \vDash H_v) \vee G_0 \nvDash \mathcal{S})\\
&\iff \exists u(G_n \vDash F_u) \Rightarrow (\exists z, v(0 \le z \le n \wedge G_z \vDash H_v) \vee \exists v(G_0 \vDash SC_v))
\end{aligned}$$

□

The idea is similar to the refinement of Lemma 5.2 (p. 75) by Lemma 6.10 (p. 123) – we establish a criterion for possible counterexamples ($G_n \vDash F_u$), then try to discard the counterexample because there is a violation of the composed guaranteed pattern ($G_z \vDash H_v$) or because the first graph of the transformation sequence is not allowed as a start graph ($G_0 \vDash SC_v$). Keep in mind that a composed start configuration pattern is defined as $\bigwedge_{o \in O} \neg SC_o$; hence $G_z \vDash H_v$ or $G_0 \vDash SC_v$ is indeed a violation of the composed start configuration pattern. Similarly, $G_z \vDash H_v$ is a violation of the composed guaranteed constraint.

However, in our symbolic approach, we will not verify specific graphs for the satisfaction or violation of (guaranteed or start configuration) graph patterns. Instead, we will use source patterns, target patterns, and target/source patterns – that is, their reduction to a graph pattern (Lemma 2.38 (p. 43)) – which represent the potentially infinite number of graphs that may occur in relevant transformation sequences. These graph patterns can then be compared to guaranteed patterns and start configuration patterns with respect to pattern implication: in particular, we can apply the theorem and algorithm used in earlier work [Dyc12] and reiterated in Theorem 6.8 and Algorithm 6.1. The result will allow us to reason about satisfiability of guaranteed and start configuration patterns by the graphs represented by the reduced source and target patterns.

Source pattern, target patterns, target/source patterns, and sequences of source/target patterns were established in Chapter 5 for the general approach. Originally, we introduced source and target patterns as unrestricted nested application conditions over a left or right rule side. Here, since our symbolic encoding (s/t-pattern sequences) needs to represent transformation sequences that lead to graph constraints of limited complexity – namely, graph patterns – we can restrict the expressive power of source patterns and target patterns – and hence, of s/t-pattern sequences – as well.

To determine appropriate restrictions on source patterns and target patterns, we analyze possible counterexamples and their symbolic encoding. As before (Chapter 5, Section 5.2, construction of s/t-pattern sequences), we will use an adjusted Seq-construction. Here, we will need to encode transformation sequences that lead to a singular pattern: in Lemmas 6.10 (p. 123) and 6.11 (p. 124), the transformation sequences' rightmost graph needs to violate a forbidden graph pattern; e.g. $G_k \vDash F_u$. By definition, $F_u = \exists(i_A : \varnothing \hookrightarrow A, ac_A)$ for a graph A and a composed negative application condition ac_A. For ease of reading, we will use $F = F_u$ for such an arbitrary pattern F_u. Of course, our algorithm would need to consider all individual patterns F_i.

Since Theorem T.1g (p. 85) of the general approach does not impose any restrictions on the type of graph constraints, we can simply consider computing $\text{Seq}^g_k(\mathcal{R}, F, \dots)$ (note: the forbidden pattern F, not the composed forbidden pattern $\mathcal{F}$) to encode all transformation sequences leading to F, i.e. all transformation sequences that need to be considered as counterexamples. Then, by step SC$_1$-1 of the construction, we will need to compute $\text{Shift}(i_R, F)$ for each right side R of rules in $\mathcal{R}$. Furthermore, since the idea is to leave out transfers of guaranteed patterns and the remaining forbidden patterns to the s/t-pattern sequences (instead taking them into account in an analysis step later on) all additional complexity in the s/t-pattern sequence can only come from the rules (including their application conditions).

We get

$$\begin{aligned}
&\mathrm{Shift}(i_R, F)\\
=&\mathrm{Shift}(i_R, \exists(i_A, ac_A))\\
=&\bigvee_{j\in J}\exists(t_j, \mathrm{Shift}(t'_j, ac_A))
\end{aligned}$$

as in the diagram below. The result is a disjunction of existential conditions over a right rule side; each existential condition has a nested application condition that is the result of shifting a composed negative application condition (ac_A) over a morphism.

$$\begin{tikzcd}
L \arrow[dr, phantom, "\triangledown ac_L" near start] & K \arrow[l, hook'] \arrow[r, hook] & R \arrow[d, hook, "t_j"'] \arrow[dr, phantom, "="] & \varnothing \arrow[l, hook', "i_R"'] \arrow[d, hook, "i_A"'] \\
 & & T_j \arrow[ul, phantom, "ac_{T_j}\triangle" near start] & A \arrow[l, hook', "t'_j"] \arrow[l, phantom, "ac_A\triangle" near start, shift left=2]
\end{tikzcd}$$

Note that this result would be a valid target pattern by Definition 5.8 of our general approach. However, it is still more complex than desirable and, indeed, necessary. Given the result's structure, we can do better: since we have a disjunction of existential conditions, each existential condition encodes one potential rule application (result) where F can be found, and hence, $\mathcal{F}$ is violated. Thus, we will (in Section 6.3) adjust the Seq-construction to split the disjunction into several target patterns, with each target pattern an existential condition over a right rule side and an appropriate nested application condition ($\mathrm{Shift}(t'_j, ac_A)$). Fortunately, the structure of composed negative application conditions is preserved by the Shift-construction: the result will again be a composed negative application condition. This has been established in Fact 6.6 (p. 119).

A generalized version of splitting graph constraints in the fashion described above (and in Lemmas 6.10 (p. 123) and 6.11 (p. 124)) could also be applied for the general approach, given suitable structure of the graph constraint in question. In particular, the nesting depth of a constraint, which is not restricted in the general approach, does not prevent a graph constraint from being split. However, arbitrary logical combinations on the first level of the constraint would cause the resulting fragments to be more complex in the general approach than in the restricted approach.

As before, target patterns will be converted to source patterns by the L-construction. Here, we have (with ac being a composed negative application condition and b the graph rule in question):

$$\begin{aligned}
&\mathrm{L}(b, \exists(t_j, ac_{T_j}))\\
=&\exists(s_j, \mathrm{L}(b', ac_{T_j})) - \text{or false}\,.
\end{aligned}$$

$$\begin{tikzcd}
L \arrow[d, hook, "s_j"] \arrow[dr, phantom, "\triangledown ac_L" near start] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] & R \arrow[d, hook, "t_j"'] \arrow[dr, phantom, "="] & \varnothing \arrow[l, hook', "i_R"'] \arrow[d, hook, "i_A"'] \\
S \arrow[ur, phantom, "\triangle ac' S_j" near start] & D_j \arrow[l, hook'] \arrow[r, hook] & T_j \arrow[ul, phantom, "ac_{T_j}\triangle" near start] & A \arrow[l, hook', "t'_j"] \arrow[l, phantom, "ac_A\triangle" near start, shift left=2]
\end{tikzcd}$$

Then, the resulting source patterns are existential conditions with a nested application condition. Again, using the L-construction on a composed negative application condition ($\mathrm{L}(b', ac)$) results in a composed negative application condition, which was established in Fact 6.7 (p. 119).

That leaves us with existential conditions with nested composed negative application conditions as source patterns and target patterns. Hence, for the restricted formal model, we refine Definitions 5.7 (p. 77) and 5.8 (p. 77) as follows:

Definition 6.12 (source pattern [3]). *Given a graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$, a* source pattern *over b is an application condition over the left side L of the form $src = \exists(s, ac_S)$ with ac_S being a composed negative application condition.*

Definition 6.13 (target pattern [3]). *Given a graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$, a* target pattern *over b is an application condition over the right side R of the form $tar = \exists(t, ac_T)$ with ac_T being a composed negative application condition.*

Instead of referring to a source pattern over a rule b, we will sometimes refer to source patterns over a left side L; the same will apply for target patterns.

As before, source and target patterns specify additional context – and, via negative application conditions, its absence – in which a rule application occur. A source pattern *src* represents all matches $m : L \hookrightarrow G$ where m satisfies *src*. Likewise, a target pattern *tar* represents all comatches $m' : R \hookrightarrow G'$ that satisfy it. Together, a source and a target pattern over the same rule can encode potential rule applications via specific matches and comatches.

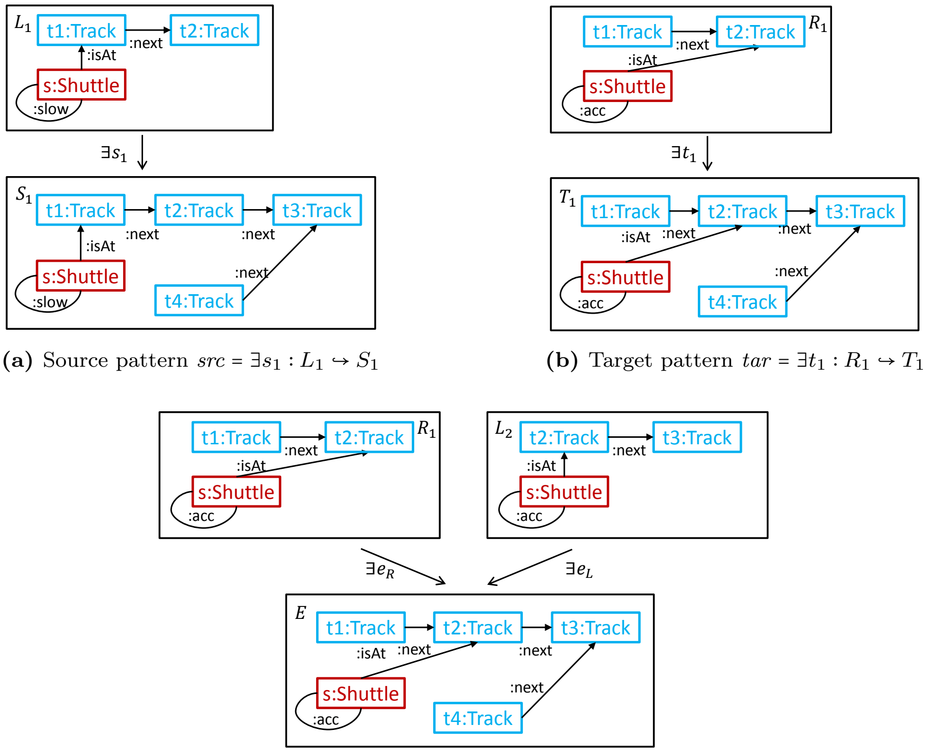
However, given our restricted formal model, we can be more specific: a source or target pattern's existential condition describes the (minimal) graph – called its (positive) context – in which the rule application (via match or comatch) may occur. It directly embeds the (co-)match in another graph – as opposed to a complex application condition whose operands and nesting structures may be difficult to grasp for a human viewer. In addition, unless the composed negative application condition is contradictory to the existential condition, it is easy to find a satisfying (co-)match by choosing the existential condition's morphism. Contradictory composed negative application are easy to spot: they contain a negative application condition whose morphism is bijective (i.e. an isomorphism).

As argued above, given our formal model, this format is sufficient to describe the context required to analyze our systems for violations of safety properties describes as composed forbidden graph patterns.

The structure for the refined target/source patterns then follows a logic similar to the definitions of source and target patterns. A target/source pattern needs to describe an overlapping of a right and a left side of two rules. As before, this can be done by using a pair of injective and jointly surjective morphisms to induce a pair of existential conditions with a common nested condition. Given the structure of source and target patterns, that nested condition will be a composed negative application condition.

Definition 6.14 (target/source pattern [3]). *Given rules $b_1 = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), ac_{L_1}, ac_{R_1} \rangle$ and $b_2 = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), ac_{L_2}, ac_{R_2} \rangle$ and a graph E with a pair of injective and jointly surjective morphisms $(e_R : R_1 \hookrightarrow E, e_L : L_2 \hookrightarrow E)$, a* target/source pattern *over (b_1, b_2) is a pair of application conditions over R_1 and L_2 of the form $(\exists(e_R, ac_E), \exists(e_L, ac_E))$ with ac_E being a composed negative application condition over E.*

A pair of morphisms with the same codomain $(m'_1 : R_1 \hookrightarrow G, m_2 : L_2 \hookrightarrow G)$ satisfies a target/source pattern (tar, src), denoted $(m'_1, m_2) \vDash (tar, src)$, if m'_1 and m_2 satisfy tar and src via a common injective morphism, i.e. if there exists $y : E \hookrightarrow G$ with $y \vDash ac_E$, $y \circ e_R = m'_1$, and $y \circ e_L = m_2$.



(a) Source pattern $src = \exists s_1 : L_1 \hookrightarrow S_1$

(b) Target pattern $tar = \exists t_1 : R_1 \hookrightarrow T_1$

(c) Target/source pattern $ts = (\exists e_R, \exists e_L)$

Figure 6.5. – Example source pattern $src = \exists s_1$, target pattern $tar = \exists t_1$, and target/source pattern $(\exists e_R, \exists e_L)$

$$\begin{tikzcd}
R_1 \arrow[dr, hook, "e_R"] \arrow[ddr, bend right, "m'_1{=}y\circ e_R"'] & & L_2 \arrow[dl, hook', "e_L"'] \arrow[ddl, bend left, "m_2{=}y\circ e_L"] \\
& E \triangleleft ac_E \arrow[d, "y"', "\models"] & \\
& G &
\end{tikzcd}$$

Example 6.15 (source, target, and target/source patterns)**.** The example source, target, and target/source patterns used in Example 5.9 (p. 77) for our general approach are also valid source, target, and target/source patterns under our restricted formal model. They are shown again in Figure 6.5. △

However, the definition of s/t-pattern sequences (Definition D.2) does not need to be refined. Since it is based on source patterns, target patterns, and target/source patterns, the changes to those concepts directly affect the structure of an s/t-pattern sequence in our restricted formal model. We reiterate that definition here:

Definition D.2 (k-sequence of source/target patterns [3])**.** *Given a $k \geq 1$, a source pattern src_1 over a rule b_1, a target pattern tar_k over a rule b_k and a number of target/source patterns (tar_i, src_{i+1}) over a number of rules b_i $(1 \leq i \leq k-1)$, $seq = src_1 \Rightarrow_{b_1} (tar_1, src_2) \Rightarrow_{b_2} \dots \Rightarrow_{b_k} tar_k$ is a* k-sequence of s/t-patterns.

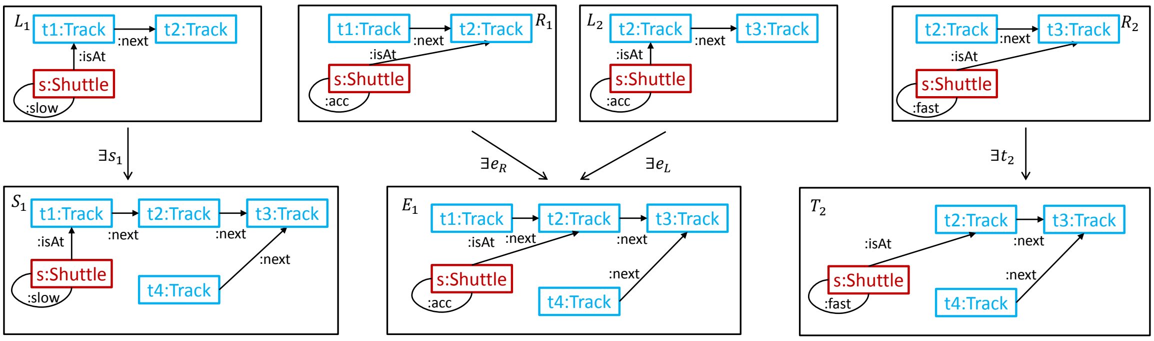


Figure 6.6. – 2-sequence of source/target patterns $seq = \exists s_1 \Rightarrow_{\mathsf{s2a}'} (\exists e_R, \exists e_L) \Rightarrow_{\mathsf{a2f}'} \exists t_2$

$$\begin{tikzcd}
src_1 \triangleright L_1 & K_1 \arrow[l, hook'] \arrow[r, hook] & R_1 \arrow[dr, "e_{R_1}"'] & & L_2 \arrow[dl, "e_{L_2}"] & \cdots & R_k \triangleleft tar_k \\
 & & & E_1 \triangleleft ac_{E_1} & & \cdots &
\end{tikzcd}$$

Satisfiability *of k-sequences of s/t-patterns is defined as follows:*

Given a sequence of transformations (of length k) trans $= G_0 \Rightarrow_{c_1,m_1,m'_1} \dots \Rightarrow_{c_k,m_k,m'_k} G_k$ and a k-sequence of s/t-patterns seq $= src_1 \Rightarrow_{b_1} (tar_1, src_2) \Rightarrow_{b_2} \dots \Rightarrow_{b_k} tar_k$, trans satisfies seq, denoted as trans $\vDash$ seq, if, for all i with $1 \le i \le k$, $c_i = b_i$, $m_i \vDash src_i$, $m'_i \vDash tar_i$ and, in particular, for all i with $1 \le i \le k-1$, $(m'_i, m_{i+1}) \vDash (tar_i, src_{i+1})$.

$$\begin{tikzcd}
src_1 \triangleright L_1 \arrow[dd, hook, "m_1 \vDash src_1"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[dd, hook] & R_1 \arrow[dr, "e_{R_1}"] \arrow[ddr, bend right, "m'_1"'] & & L_2 \arrow[dl, "e_{L_2}"'] \arrow[ddl, bend left, "m_2"] & \cdots & R_k \triangleleft tar_k \arrow[dd, hook, "m'_k \vDash tar_k"] \\
 & & = & E_1 \triangleleft ac_{E_1} \arrow[d, hook, "y_1 \vDash ac_{E_1}"] & = & \cdots & \\
G_0 & D_1 \arrow[l, hook'] \arrow[rr, hook] & & G_1 & & \cdots & G_k
\end{tikzcd}$$

Two k-sequences of s/t-patterns seq, seq$'$ are equivalent *(seq $\equiv$ seq$'$), if for all transformation sequences trans it holds that trans $\vDash$ seq $\Leftrightarrow$ trans $\vDash$ seq$'$.*

Example 6.16 (s/t-pattern sequence)**.** The s/t-pattern sequence $\exists s_1 \Rightarrow_{\mathsf{s2a}'} (\exists e_R, \exists e_L) \Rightarrow_{\mathsf{a2f}'} \exists t_2$ shown in Example 5.12 (p. 82) also conforms to our restricted formal model; it is shown again in Figure 6.6. △

This concludes the introduction of our symbolic encoding to represent transformation sequences following the requirements of our restricted formal model. The following section will focus on the construction of specific s/t-pattern sequences required in our verification approach.

6.3. Construction of Pattern Sequences

Given our restricted formal model and the refinements to source patterns, target patterns, target/source patterns, and, implicitly, s/t-pattern sequences, we now refine the Seq-construction (Theorem T.1g (p. 85)) of our general approach. In the restricted approach, we defer the question of constraint satisfaction by intermediate graphs of satisfying transformation sequences; this is an attempt to reduce complexity and improve performance. Thus, the construction only takes a set of graph rules ($\mathcal{R}$) and a (forbidden) graph pattern (C) as a parameter. It then creates s/t-pattern sequences whose satisfying transformation sequences use rules in $\mathcal{R}$ and lead to C.

In Lemmas 6.10 (p. 123) and 6.11 (p. 124), we have established the structure of transformation sequences that serve as counterexamples for k-inductive invariants and validity of a composed forbidden pattern in $k{-}1$-bounded state spaces (under a composed pattern $\mathcal{H}$). In both cases, sequences lead to a forbidden pattern F, which is part of a composed forbidden pattern $\mathcal{F}$. Hence, for both cases, we will apply the refined Seq-construction with the parameters $\mathcal{R}$ and F – for each forbidden pattern F contained in $\mathcal{F}$. Both cases, including the subsequent analysis step that also takes $\mathcal{H}$ and $\mathcal{S}$ into account, will be explained in Sections 6.4 and 6.5.

Theorem T.1r (construction of s/t-pattern sequences [3])**.** *There is a construction* Seq_k^r *such that for each graph pattern* $C = \exists(i_P, ac_P)$*, rule set* $\mathcal{R}$*, and* $k \geq 1$*,* $\mathrm{Seq}_k^r(\mathcal{R}, C)$ *is a set of* k*-sequences of* s/t*-patterns such that:*

1. *For each transformation sequence trans to* $\mathcal{R}$ *and of length* k *leading to* C*, there exists a seq* $\in \mathrm{Seq}_k^r(\mathcal{R}, C)$ *such that trans* $\models$ *seq.*
2. *Given a seq* $\in \mathrm{Seq}_k^r(\mathcal{R}, C)$*, for every transformation sequence trans with trans* $\models$ *seq, trans leads to* C*.*

Construction. Seq_k^r *is inductively constructed as follows (with appropriate indexes and index sets), starting with* Seq_1^r *(left figure), which consists of five steps SC*$_1$*-1 to SC*$_1$*-5:*

SC$_1$*-1: For each rule* $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$*,* $\mathrm{Shift}(i_R, C) = \bigvee_{j \in J_b} tar_{b,j}$ *is a disjunction of target patterns over* R *of the form* $tar_{b,j} = \exists(t_j, ac_{T_j})$*.*

SC$_1$*-2: For each such target pattern* $tar_{b,j}$*,* $src'_{b,j} = \mathrm{L}(b, tar_{b,j})$ *is a source pattern over* L *of the form* $src'_{b,j} = \mathrm{false}$ *or* $src'_{b,j} = \exists(s_j, ac'_{S_j})$*.*

SC$_1$*-3: For the latter case,* $src_{b,j} = \exists(s_j, ac_{S_j})$ *with* $ac_{S_j} = ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b))$ *is a source pattern.*

SC$_1$*-4: For each such pair of source and target pattern* $src_{b,j}$ *and* $tar_{b,j}$*,* $src_{b,j} \Rightarrow_b tar_{b,j}$ *is a* 1*-sequence of* s/t*-patterns.*

SC$_1$*-5: Finally, we define* $\mathrm{Seq}_1^r(\mathcal{R}, C) = \{src_{b,j} \Rightarrow_b tar_{b,j} \mid b \in \mathcal{R} \wedge j \in J_b\}$ *as the set of these sequences.*

$$\begin{tikzcd}
L \arrow[d, hook, "s_j"] \arrow[dr, phantom, "\triangledown ac_L" near start] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] & R \arrow[d, hook, "t_j"'] \arrow[dr, phantom, "="] & \varnothing \arrow[l, hook', "i_R"'] \arrow[d, hook, "i_P"'] \\
S_j \arrow[ur, phantom, "\triangle ac'_{S_j}" near start] & D_j \arrow[l, hook'] \arrow[r, hook] & T_j \arrow[ul, phantom, "ac_{T_j}\triangle" near start] & P \arrow[l, hook', "p_j"] \arrow[ul, phantom, "ac_P\triangle" near start]
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d, hook, "s_j"] \arrow[dr, phantom, "\triangledown ac_L" near start] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] & R \arrow[d, hook, "t_j"'] & L_1 \arrow[d, hook, "s_1"] \\
S_j \arrow[ur, phantom, "\triangle ac'_{S_j}" near start] & D_j \arrow[l, hook'] \arrow[r, hook] & T_j \arrow[ul, phantom, "ac_{T_j}\triangle" near start] & S_1 \arrow[l, hook', "s'_j"'] \arrow[ul, phantom, "ac_{S_1}\triangle" near start]
\end{tikzcd}$$

Given $\mathrm{Seq}_k^r(\mathcal{R}, F)$*, we construct* $\mathrm{Seq}_{k+1}^r(\mathcal{R}, F)$ *as follows (right figure).*

SC$_k$*-1: For each sequence seq* $= src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k \in \mathrm{Seq}_k^r(\mathcal{R}, C)$ *with* $src_1 = \exists(s_1 : L_1 \hookrightarrow S_1, ac_{S_1})$*, each* $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$*, and each graph* T_j *and pair of injective and jointly surjective morphisms* $(t_j : R \hookrightarrow T_j, s'_j : S_1 \hookrightarrow T_j)$*,* $tar_{b,j} = \exists(t_j, ac_{T_j})$ *with* $ac_{T_j} = \mathrm{Shift}(s'_j, ac_{S_1})$ *is a target pattern over* R*.*

SC$_k$*-1*$^+$*: For each such target pattern* $tar_{b,j}$*,* $src^+_{1,j} = \exists(s'_j \circ s_1, ac_{T_j})$ *is a source pattern over* L_1 *and* $(tar_{b,j}, src^+_{1,j})$ *is a target/source pattern over* (b, b_1)*.*

SC$_k$*-2: For each such target pattern* $tar_{b,j}$*,* $src'_{b,j} = \mathrm{L}(b, tar_{b,j})$ *is a source pattern over* L *of the form* $src'_{b,j} = \mathrm{false}$ *or* $src'_{b,j} = \exists(s_j, ac'_{S_j})$*.*

SC$_k$*-3: For the latter case,* $src_{b,j} = \exists(s_j, ac_{S_j})$ *with* $ac_{S_j} = ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b))$ *is a source pattern over* L*.*

SC_k-4: For each such pair of source and target pattern $src_{b,j}$ and $tar_{b,j}$, $src_{b,j} \Rightarrow_b tar_{b,j}, src_1^+ \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ is a k+1-sequence of s/t-patterns.

SC_k-5: Finally, we define $\mathrm{Seq}^r_{k+1}(\mathcal{R}, C) = \{src_{b,j} \Rightarrow_b tar_{b,j}, src^+_{1,j} \Rightarrow \dots \Rightarrow tar_k \mid b \in \mathcal{R} \wedge j \in J_b \wedge seq \in \mathrm{Seq}^r_k(\mathcal{R}, C)\}$ as the set of these sequences.

Also, given a set of rules $\mathcal{R}$ and a composed graph pattern $\mathcal{C} = \bigwedge_{i \in I} \neg C_i$ with graph patterns C_i, we define $\mathrm{Seq}^r_k(\mathcal{R}, \neg\mathcal{C}) = \bigcup_{i \in I} \mathrm{Seq}^r_k(\mathcal{R}, C_i)$ and $\mathrm{SEQ}^r_k(\mathcal{R}, \neg\mathcal{C}) = \bigcup_{1 \le j \le k} \mathrm{Seq}^r_j(\mathcal{R}, \neg\mathcal{C})$.

Proof. *1.* First, we will prove (1) by induction.

$$\begin{tikzcd}
& L_1 \arrow[d, hook, "s_j"] \arrow[ddl, bend right, "m_1 \vDash ac_{L_1}"'] \arrow[dr, phantom, "\triangledown ac_{L_1}"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dd, hook, bend left] & R_1 \arrow[d, hook, "t_j"'] \arrow[ddr, bend left, "m'_1"] & \varnothing \arrow[l, hook', "i_{R_1}"'] \arrow[d, hook, "i_P"] \arrow[dd, hook, bend left, "i_{G_1} \vDash \exists(i_P, ac_P)"] \arrow[dl, phantom, "="] \\
\arrow[r, phantom, "="] & S_j \arrow[dl, hook, "q \vDash ac'_S"'] \arrow[ur, phantom, "\triangle ac'_{S_j}"] & D_j \arrow[l, hook'] \arrow[r, hook] \arrow[ur, phantom, "ac_{T_j} \triangle"] & T_j \arrow[ur, phantom, "ac_P \triangle"] & P \arrow[l, hook', "p_j"] \\
G_0 & & K'_1 \arrow[ll, hook'] \arrow[rr, hook] & & G_1
\end{tikzcd}$$

Base case. Let $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} G_1$ be a transformation sequence leading to $C = \exists(i_P : \varnothing \hookrightarrow P, ac_P)$, i.e. $G_1 \vDash C$. Hence, $i_{G_1} \vDash C$ and with $m'_1 \circ i_{R_1} = i_{G_1}$, we get $m'_1 \circ i_{R_1} \vDash C$. By the Shift-lemma, we have $m'_1 \vDash \mathrm{Shift}(i_{R_1}, C)$ and, considering the Seq-construction, $m'_1 \vDash tar_{b_1,j}$ for a specific $j \in J_{b_1}$ (and $tar_{b_1,j} = \exists(t_j, ac_{T_j})$).

$G_0 \Rightarrow_{b_1,m_1,m'_1} G_1$ implies $m_1 : L_1 \hookrightarrow G_0$ with $m_1 \vDash ac_{L_1} \wedge \mathrm{Appl}(b_1)$. With $m'_1 \vDash tar_{b_1,j}$ and the L-construction, we have $m_1 \vDash \mathrm{L}(b_1, tar_{1,j})$ and, considering the construction above, $m_1 \vDash src'_{b_1,j}$ (where $src'_{b_1,j} = \exists(s_j, ac'_{S_j})$). More specifically, there exists a monomorphism $q : S_j \hookrightarrow G_0$ such that $q \circ s_j = m_1$ and $q \vDash ac'_{S_j}$. Because of $m_1 \vDash ac_{L_1} \wedge \mathrm{Appl}(b_1)$ and the Shift-construction, $q \circ s_j \vDash ac_{L_1} \wedge \mathrm{Appl}(b_1)$ yields $q \vDash \mathrm{Shift}(s_j, ac_{L_1} \wedge \mathrm{Appl}(b_1))$, which leads to $q \vDash ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_{L_1} \wedge \mathrm{Appl}(b_1))$ and $m_1 \vDash src_{b_1,j}$ with $src_{b_1,j} = \exists(s_j, ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_{L_1} \wedge \mathrm{Appl}(b_1)))$. Then, $seq = src_{b_1,j} \Rightarrow_{b_1} tar_{b_1,j} \in \mathrm{Seq}^r_1(\mathcal{R}, F)$ and with $m'_1 \vDash tar_{b_1,j}$ and $m_1 \vDash src_{b_1,j}$, we have $trans \vDash seq$.

$$\begin{tikzcd}
& L \arrow[d, hook, "s"] \arrow[ddl, bend right, "m \vDash ac_L"'] \arrow[dr, phantom, "\triangledown ac_L"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dd, hook, bend left] & R \arrow[d, hook, "t"'] \arrow[ddr, bend left, "m'"'] & L_1 \arrow[d, hook, "s_1"'] \arrow[dd, hook, bend left, "m_1 \vDash \exists(s_1, ac_{S_1}) = src_1"] \\
\arrow[r, phantom, "="] & S \arrow[dl, hook, "q \vDash ac'_S"'] \arrow[ur, phantom, "\triangle ac'_S"] & D_j \arrow[l, hook'] \arrow[r, hook] \arrow[ur, phantom, "ac_T \triangle"] & T \arrow[dr, hook, bend right, "y'"'] \arrow[ur, phantom, "ac_{S_1} \triangle"] & S_1 \arrow[l, hook', "s'"] \arrow[d, hook, "y"] \\
G & & K' \arrow[ll, hook'] \arrow[rr, hook] & & G_0
\end{tikzcd}$$

Inductive step. Let $\mathrm{Seq}^r_k(\mathcal{R}, C)$ be a set of sequences such that for each k-sequence of transformations *trans* that leads to F, there is a k-sequence of s/t-patterns $seq \in \mathrm{Seq}^r_k(\mathcal{R}, C)$ such that $trans \vDash seq$.

Consider a k+1-sequence of transformations $trans' = G \Rightarrow_{b,m,m'} G_0 \Rightarrow_{b_1,m_1,m'_1} \cdots \Rightarrow_{b_k,m_k,m'_k} G_k$ that leads to C. Then, $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \cdots \Rightarrow_{b_k,m_k,m'_k} G_k$ is a k-sequence of transformations that leads to C. By assumption, there is a k-sequence of s/t-patterns $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k \in \mathrm{Seq}^r_k(\mathcal{R}, C)$ such that $trans \vDash seq$ and $src_1 = \exists(s_1, ac_{S_1})$.

Since $trans \vDash seq$ (and with $G_0 \Rightarrow_{b_1,m_1,m'_1} G_1$), we have a match $m_1 : L_1 \hookrightarrow G_0$ with $m_1 \vDash src_1$, implying the existence of an injective morphism $y : S_1 \hookrightarrow G_0$ such that $y \circ s_1 = m_1$ and $y \vDash ac_{S_1}$. Since there is a transformation $G \Rightarrow_{b,m,m'} G_0$, there is a comatch $m' : R \hookrightarrow G_0$. Then, given

$m' : R \hookrightarrow G_0$ and $y : S_1 \hookrightarrow G_0$ and by $\mathcal{E}'$-$\mathcal{M}$-pair factorization [EGH⁺14], there is a graph T with a pair of jointly surjective morphisms $(t : R \hookrightarrow T, s' : S_1 \hookrightarrow T)$ such that there exists an injective morphism $y' : T \hookrightarrow G_0$ with $y' \circ t = m'$ and $y' \circ s' = y$. (By decomposition, t and s' are injective.)

By construction of $\mathrm{Seq}^r_{k+1}(\mathcal{R}, C)$, for that particular graph T and pair of injective and jointly surjective morphisms $(t : R \hookrightarrow T, s' : S_1 \hookrightarrow T)$, there is an s/t-pattern sequence $\mathit{seq}' \in \mathrm{Seq}^r_{k+1}(\mathcal{R}, C)$ with $\mathit{seq}' = \mathit{src} \Rightarrow_b (\mathit{tar}, \mathit{src}^+_1) \Rightarrow_{b_1} \dots \Rightarrow_{b_k} \mathit{tar}_k$ where $(\mathit{tar}, \mathit{src}^+_1) = (\exists(t, \mathit{ac}_T), \exists(s' \circ s_1, \mathit{ac}_T))$ is a target/source pattern with $\mathit{ac}_T = \mathrm{Shift}(s', \mathit{ac}_{S_1})$. Furthermore, by construction, $\mathit{src} = \exists(s, \mathit{ac}_S)$ where $\mathit{ac}_S = \mathit{ac}'_S \wedge \mathrm{Shift}(s, \mathit{ac}_L) \wedge \mathrm{Shift}(s, \mathrm{Appl}(b))$ and $\exists(s, \mathit{ac}'_S) = \mathrm{L}(b, \mathit{tar}) = \mathrm{L}(b, \exists(t, \mathit{ac}_T))$. We will show that trans' satisfies seq'.

Since $\mathit{ac}_T = \mathrm{Shift}(s', \mathit{ac}_{S_1})$ and $y' \circ s' = y$, we have $y' \vDash \mathit{ac}_T$. With $y' \circ t = m'$ and $y' \circ s' \circ s_1 = y \circ s_1 = m_1$, we have $m' \vDash \mathit{tar}$ and $m_1 \vDash \mathit{src}^+_1$ via the common injective morphism y' and hence, (m', m_1) satisfies the target/source pattern $(\mathit{tar}, \mathit{src}^+_1)$.

Furthermore, $\mathit{src}' = \mathrm{L}(b, \mathit{tar})$ and, by the L-construction and with $G \Rightarrow_{b,m,m'} G_0$, we have $m \vDash \mathit{src}'$. In particular, $\mathit{src}' = \mathrm{L}(b, \exists(t, \mathit{ac}_T)) = \exists(s, \mathit{ac}'_S)$; since $m \vDash \mathit{src}'$, src' cannot be false. Then, there is an injective morphism $q : S \hookrightarrow G$ such that $q \circ s = m$ and $q \vDash \mathit{ac}'_S$. $G \Rightarrow_{b,m,m'} G_0$ implies $m \vDash \mathit{ac}_L \wedge \mathrm{Appl}(b)$ and with $q \circ s = m$, we have $q \vDash \mathrm{Shift}(s, \mathit{ac}_L \wedge \mathrm{Appl}(b))$. Thus, $q \vDash \mathit{ac}'_S \wedge \mathrm{Shift}(s, \mathit{ac}_L \wedge \mathrm{Appl}(b))$, which gives us $q \vDash \mathit{ac}_S$ and hence, $m \vDash \mathit{src}$.

Finally, $m \vDash \mathit{src}$, $(m', m_1) \vDash (\mathit{tar}, \mathit{src}^+_1)$, and $\mathit{trans} \vDash \mathit{seq}$ imply $\mathit{trans}' \vDash \mathit{seq}'$, concluding the inductive proof.

2. Second, we will prove (2) by induction:

Base case. Let $\mathit{seq} = \mathit{src}_1 \Rightarrow_{b_1} \mathit{tar}_1$ be an arbitrary 1-sequence of s/t-patterns such that $\mathit{seq} \in \mathrm{Seq}^r_1(\mathcal{R}, C)$ with $b_1 \in \mathcal{R}$ and $C = \exists(i_P, \mathit{ac}_P)$. Consider an arbitrary transformation sequence $\mathit{trans} = G_0 \Rightarrow_{b_1,m_1,m'_1} G_1$ such that $\mathit{trans} \vDash \mathit{seq}$. Hence, we have $m'_1 \vDash \mathit{tar}_1$ and, by construction, $m'_1 \vDash \mathrm{Shift}(i_{R_1}, C)$. Then, $m'_1 \circ i_{R_1} \vDash C$ and with $i_{G_1} : \varnothing \hookrightarrow G_1$ and $i_{G_1} = m'_1 \circ i_{R_1}$, we gain $i_{G_1} \vDash C$, implying $G_1 \vDash C$ and thus, trans leads to C. Consequently, for every s/t-pattern sequence seq in $\mathit{Seq}_1(\mathcal{R}, C)$, every transformation sequence trans with $\mathit{trans} \vDash \mathit{seq}$ leads to C.

$$\begin{tikzcd}
 & L_1 \arrow[d, hook, "s_j"] \arrow[dl, phantom, ""] \arrow[ddl, hook, bend right, "m_1 \vDash ac_{L_1}"'] \arrow[dr, phantom, "\triangledown ac_{L_1}" near start] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dd, hook, bend left] & R_1 \arrow[d, hook, "t_j"'] \arrow[ddr, hook, bend left, "m'_1" near end] \arrow[dr, phantom, "="] & \varnothing \arrow[l, hook', "i_{R_1}"'] \arrow[d, hook, "i_P"'] \arrow[dd, hook, bend left, "i_{G_1} \vDash \exists(i_P, ac_P)"] \\
 & S_j \arrow[ur, phantom, "\vartriangle ac'_{S_j}" near start] & D_j \arrow[l, hook'] \arrow[r, hook] \arrow[ur, phantom, "ac_{T_j} \vartriangle" near end] & T_j & P \arrow[l, hook', "p_j"] \arrow[ul, phantom, "ac_P \vartriangle" near start] \\
G_0 & & K'_1 \arrow[ll, hook'] \arrow[rr, hook] & & G_1
\end{tikzcd}$$

Inductive step. Let $\mathrm{Seq}^r_k(\mathcal{R}, C)$ be a set of s/t-pattern sequences such that for every s/t-pattern sequence in $\mathrm{Seq}^r_k(\mathcal{R}, C)$, each transformation sequence trans with $\mathit{trans} \vDash \mathit{seq}$ leads to C.

Consider $\mathrm{Seq}^r_{k+1}(\mathcal{R}, C)$ and $\mathit{seq}' = \mathit{src}_0 \Rightarrow_{b_0} \mathit{tar}_0, \mathit{src}^+_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} \mathit{tar}_k$ as an arbitrary $k{+}1$-sequence of s/t-patterns with $\mathit{seq}' \in \mathrm{Seq}^r_{k+1}(\mathcal{R}, C)$. Consider an arbitrary transformation sequence $\mathit{trans}' = G'_0 \Rightarrow_{b_0,m_0,m'_0} G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ such that $\mathit{trans}' \vDash \mathit{seq}'$. By construction, there is a k-sequence of s/t-patterns $\mathit{seq} = \mathit{src}_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} \mathit{tar}_k$ with $\mathit{seq} \in \mathrm{Seq}^r_k(\mathcal{R}, C)$ and for $\mathit{trans} = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$, we have $\mathit{trans} \vDash \mathit{seq}$. By assumption, trans leads to C and hence, $G_k \vDash C$. Consequently, trans' leads to C, which concludes the inductive proof. □

Again, the Seq-construction comes with two properties: One, all transformation sequences leading to the pattern C will be represented by one of the constructed s/t-pattern sequences.

Two, every transformation sequence satisfying an s/t-pattern sequence created by the construction will lead to the pattern C. Similar to the general version of the Seq-construction, we get a finite set of s/t-pattern sequences from this construction.

Verifying a composed forbidden pattern $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$ as k-inductive invariant then involves computing $\mathrm{Seq}^r_k(\mathcal{R}, \neg\mathcal{F}) = \bigcup_{i\in I} \mathrm{Seq}^r_k(\mathcal{R}, F_i)$. Verifiying the validity of $\mathcal{F}$ in $k{-}1$-bounded state spaces involves computation of $\mathrm{SEQ}^r_{k-1}(\mathcal{R}, \neg\mathcal{F})$. In both cases, the s/t-pattern sequences have to be analyzed in subsequent steps.

As before, we can also express the Seq-construction in a declarative fashion rather than stepwise:

$$\mathrm{Seq}^r_1(\mathcal{R}, C) = \{src_{b,j} \Rightarrow_b tar_{b,j} \mid b \in \mathcal{R} \wedge j \in J_b\} \tag{SC$_1$-4/5}$$

where, given $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle$,

$$src_{b,j} = \exists(s_j, ac_{S_j}) \text{ with } ac_{S_j} = ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b)), \tag{SC$_1$-3}$$

$$\exists(s_j, ac'_{S_j}) = src'_{b,j} = \mathrm{L}(b, tar_{b,j}), \text{ and} \tag{SC$_1$-2}$$

$$tar_{b,j} = \exists(t_j, ac_{T_j}) \text{ where } \bigvee_{j\in J_b} tar_{b,j} = \mathrm{Shift}(i_R, \mathcal{C}). \tag{SC$_1$-1}$$

For the computation of $\mathrm{Seq}^r_{k+1}(\mathcal{R}, C)$, given $\mathrm{Seq}^r_k(\mathcal{R}, C)$, we have:

$$\mathrm{Seq}^r_{k+1}(\mathcal{R}, C) = \{src_{b,j} \Rightarrow_b tar_{b,j}, src^+_{1,j} \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k \mid b \in \mathcal{R} \wedge j \in J_b \wedge seq \in \mathrm{Seq}^r_k(\mathcal{R}, C)\} \tag{SC$_k$-4/5}$$

where, given $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle$, $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$ with $src_1 = \exists(s_1, ac_{S_1})$, and considering all pairs of injective and jointly surjective morphism pairs $(t_j : R \hookrightarrow T_j, s'_j : S_1 \hookrightarrow T_j)$,

$$src_{b,j} = \exists(s_j, ac_{S_j}) \text{ with } ac_{S_j} = ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b)), \tag{SC$_1$-3}$$

$$\exists(s_j, ac'_{S_j}) = src'_{b,j} = \mathrm{L}(b, tar_{b,j}), \tag{SC$_1$-2}$$

$$src^+_{1,j} = \exists(s'_j \circ s_1, ac_{T_j}), \text{ and} \tag{SC$_k$-1$^+$}$$

$$tar_{b,j} = \exists(t_j, ac_{T_j}) \text{ with } ac_{T_j} = \mathrm{Shift}(s'_j, ac_{S_1}). \tag{SC$_1$-1}$$

The construction will be executed from right to left, starting with the case for length 1 (steps SC$_1$-1 to SC$_1$-5) and then iteratively applying the steps for prolonging the k-sequences to $k + 1$ (steps SC$_k$-1 to SC$_k$-5). In short, the steps of the Seq-construction have the following effect:

SC$_1$-1 creates combinations of the graph pattern C with right rule sides for all rules in $\mathcal{R}$, thereby spawning a number of target patterns for each rule.

SC$_1$-2 creates source patterns by shifting all target patterns to the respective left rule side. Intuitively, rules are applied in reverse direction.

SC$_1$-3 adds the rules' left application condition and applicability condition to the source pattern to ensure correct rule application in satisfying transformation sequences.

SC$_1$-4/5 combines the pairs of source and target patterns into a number of s/t-pattern sequences of length 1.

SC$_k$-1 creates combinations of the leftmost source pattern and the right rule sides for all s/t-pattern sequences in $\mathrm{Seq}^r_k(\mathcal{R}, C)$ and rules in $\mathcal{R}$, thereby spawning a number of target patterns for each pair of a rule and a sequence.

SC$_k$-1$^+$ makes sure there is equivalent information in the newly created target patterns and the (leftmost) source patterns used in their creation and combines them in target/source patterns.

SC$_k$-2 creates source patterns by shifting all target patterns to the respective left rule side. Intuitively, rules are applied in reverse direction.

SC$_k$-3 adds the rules' left application condition and applicability condition to the source pattern to ensure correct rule application in satisfying transformation sequences.

SC$_k$-4/5 combines the constructed source patterns and target/source patterns with their corresponding sequences in $\mathrm{Seq}_k^r(\mathcal{R}, C)$ to form a number of s/t-pattern sequences of length $k + 1$.

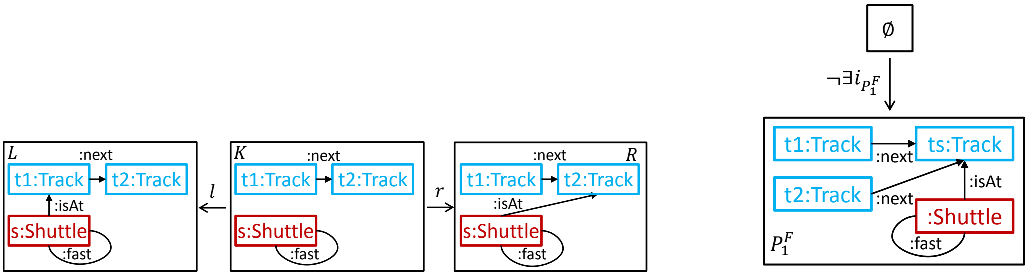
Note that there are several important differences to the Seq-construction used in the general approach (Theorem T.1g). We will explain those differences before delving into the details of the construction steps:

- $\mathrm{Seq}_k^r(\mathcal{R}, C)$ takes a set of graph rules and a graph pattern as parameters, as opposed to a set of two graph rules and two graph constraints of arbitrary complexity ($\mathrm{Seq}_k^g(\mathcal{R}, \mathcal{C}_1, \mathcal{C}_2)$).
- There is no graph constraint to be satisfied by intermediate graphs of satisfying transformation sequences; hence, steps SC$_1$-3 and SC$_\mathrm{k}$-3 only transfer the left application condition to the context of the source pattern.
- The construction $\mathrm{Shift}(i_R, C)$ (step SC$_1$-1) and its equivalent in step SC$_\mathrm{k}$-1 are not used to produce one target pattern over the respective right rule side, but a number of target patterns, each of which spawns its own s/t-pattern sequence.
- In steps SC$_1$-2 and SC$_\mathrm{k}$-2, L-constructions with the direct result false are identified and discarded.

The first (and second) difference is owed to our goal of reducing the complexity of the construction and its results. Since we aim to shift the consideration of guaranteed constraints from the construction of s/t-pattern sequences to the analysis of said sequences, a constraint that should be satisfied by intermediate graphs in satisfying transformation sequences is not part of the construction here. Thus, the two properties ensured by the construction only refer to the notion of leading to a constraint, which is passed as the parameter C. The operation of shifting constraints is costly not only in itself, but also spawns subsequent computations when its results are again transferred to the context of right rule sides or over reverse rule applications.

While systems following our restricted formal model are equipped with a composed forbidden pattern, the construction's parameter C may only be a graph pattern. This follows the notion introduced in Lemmas 6.10 (p. 123) and 6.11 (p. 124): due to the specific structure of a composed forbidden pattern, it is sufficient for a counterexample (transformation sequence) to lead to one of the forbidden patterns F_i. Hence, our construction focuses on individual patterns; analyzing all forbidden patterns via the Seq-construction and subsequent analysis steps in their entirety then amounts to analyzing the composed forbidden pattern. Formally, this is described by the definition $\mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F}) = \bigcup_{i \in I} \mathrm{Seq}_k^r(\mathcal{R}, F_i)$. Furthermore, note that given a pattern F, $\mathrm{Seq}_k^g(\mathcal{R}, F, \mathrm{true})$ and $\mathrm{Seq}_k^r(\mathcal{R}, F)$ are equivalent in the sense that its sets of satisfying transformation sequences are equal. Given a composed forbidden pattern $\mathcal{F}$, the same holds for $\mathrm{Seq}_k^g(\mathcal{R}, \neg\mathcal{F}, \mathrm{true})$ and $\mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$.

The third point makes s/t-pattern sequences more specific with respect to the transformation sequences represented. Since C is a graph pattern, the result of $\mathrm{Shift}(i_R, C)$ will always be a disjunction of existential conditions with nested composed negative application conditions – $\bigvee_{j \in J_b} \exists(t_j, ac_{T_j})$. Creating a new target pattern – and, potentially, a new s/t-pattern sequence –



(a) Graph rule f2f′

(b) $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

Figure 6.7. – Graph rule f2f′ and composed forbidden pattern $\mathcal{F} = \neg F = \neg\exists i_{P_1^F}$

for each existential condition in the disjunction reduces the number of transformation sequences satisfying each s/t-pattern sequence, although it will often still be infinite. The desired effect is threefold: First, such an s/t-pattern sequence will provide a more specific and concise view of the system's behavior leading to C. Second, it is easier to analyze and potentially discard individual sequences. Finally, splitting sequences may allow for parallelization of the analysis. Since all target patterns appear as part of a disjunction in the result, any one of the newly created target patterns and s/t-pattern sequences fulfill the properties defined in the theorem and have to be considered. Also, recall that our definition of target patterns (Definition 6.13 (p. 127)) was chosen with the specific intention of splitting such disjunctions: a target pattern in the restricted formal model can only be an existential condition with a nested composed negative application condition, not a disjunction of such conditions.

The last point is also rooted in the desire to simplify the contructions and their results. In the general approach, the L-construction may have complex conditions as input; then, the result's equivalence to false can be costly to determine. Here, calculating $\mathrm{L}(b, tar)$ for a rule b and a target pattern tar is not nearly as challenging with respect to computational effort. Since a target pattern is an existential condition with a nested composed negative application condition $(\exists(t_j, ac_{T_j}))$, $\mathrm{L}(b, tar)$ only has the direct result false if the pushout complement to $K \hookrightarrow R \hookrightarrow T_j$ does not exist. Then, continuing the calculation for that s/t-pattern sequence is unnecessary – since false cannot be satisfied by any morphism, the final s/t-pattern sequence cannot be satisfied by any transformation sequence.

If said pushout complement does exist, the result may still be unsatisfiable: if one of the transferred negative application conditions consists of an isomorphism in its (negated) existential condition, the source pattern is again unsatisfiable by any morphism. Since this consideration is more involved from a formal perspective, this aspect is not included in the Seq-construction's definition. However, both aspects are part of the implementation: discarding unsatisfiable s/t-pattern sequences earlier reduces the computational effort of both the construction of sequences and their analysis.

Example 6.17 (Seq-construction, example system)**.** In order to keep this example short, we will again use a minimal system, consisting of the graph rule $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ (hence, $\mathcal{R} = \{\mathsf{f2f'}\}$) and a composed forbidden pattern $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$. The rule and the graph pattern are shown in Figures 6.7(a) and 6.7(b), respectively; they are unchanged in comparison to Example 6.1 (p. 111). We will compute $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$, which would be appropriate in order to determine whether $\mathcal{F}$ is a 2-inductive invariant for $GTS = (TG, \mathcal{R})$.

As in the corresponding example for the general approach, we will distinguish between $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ and $\mathsf{f2f'} = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), \mathrm{true}, \mathrm{true}\rangle$. The former will refer to the appearance of rule f2f′ in the context of steps $\mathrm{SC_1}$-1 to $\mathrm{SC_1}$-5 and the latter to its appearance in the context of steps $\mathrm{SC_k}$-1 to $\mathrm{SC_k}$-5, although the rules' contents are

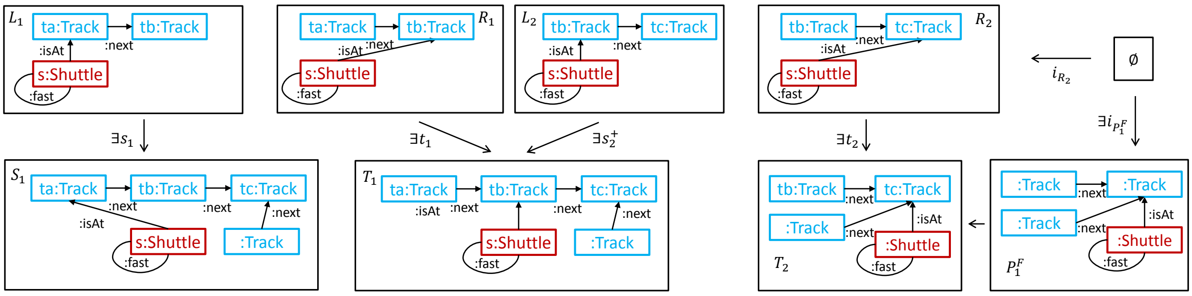


Figure 6.8. – S/t-pattern sequence $seq_2 = src_1 \Rightarrow_{\mathsf{f2f'}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f'}} tar_2$ with $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$

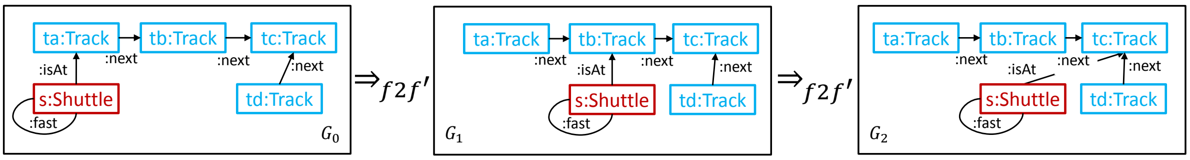


Figure 6.9. – Transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f'},m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f'},m_2,m_2'} G_2$; $trans \models seq_2$

identical. Since the rule does not delete any nodes, it has a trivial applicability condition $\mathrm{Appl}(\mathsf{f2f'}) = \mathrm{true}$.

Figure 6.8 shows one s/t-pattern sequence (of length 2) seq_2 that is contained in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$. The construction and origin of its individual parts are explained in Examples B.2-B.11 in Appendix B; here, we will only provide a short overview.

In particular,

$$\begin{aligned} seq_2 =& src_1 \Rightarrow_{\mathsf{f2f'}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f'}} tar_2 \\ =& \exists s_1 \Rightarrow_{\mathsf{f2f'}} (\exists t_1, \exists s_2^+) \Rightarrow_{\mathsf{f2f'}} \exists t_2, \end{aligned}$$

where the steps, their computations, and the corresponding examples and figures for this particular s/t-pattern sequence are listed in Table 6.2. Intuitively, the result can be explained as follows: the existential condition $\exists t_2$ describes one possibility where the application of rule $\mathsf{f2f'}$ has lead to a shuttle driving fast on a switch; then, reverse application of the rule via the L-construction determines the situation before that rule application. The resulting situation is described by an existential condition whose context is again combined with the right side of rule $\mathsf{f2f'}$. One such overlapping results in the target pattern $\exists t_1$ – and reverse application of the rule gives us $\exists s_1$, all of which are part of seq_2.

We can see already that the sequence is more specific than its counterpart in Example 5.13 (p. 90) for the general case: the source, target/source, and target patterns $\exists s_1$, $(\exists t_1, \exists s_2^+)$, and $\exists t_2$, are simple existential conditions. Of course, $\mathrm{Seq}_2^r(\mathcal{R}, F)$ will contain more than just that one s/t-pattern sequence seq_2 – and will necessarily contain more sequences than $\mathrm{Seq}_2^g(\mathcal{R}, F_1, \mathrm{true})$.

Note that the Seq-construction does not provide a means to find a satisfying transformation sequence or even to establish its existence. For the sake of this example, such a satisfying transformation sequence is shown in Figure 6.9. By Theorem T.1r (p. 130), we know that given $trans = G_0 \Rightarrow_{\mathsf{f2f'},m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f'},m_2,m_2'} G_2$ with $trans \models seq_2$ and $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$, we have $G_2 \models F_1$. Indeed, G_2 contains a violation of this forbidden pattern – a shuttle in speed mode fast on a switch. △

Table 6.2. – Computation steps of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$

step	computation	Figure	Example
SC_1-1	$tar_2 = \exists t_2$ (and $\bigvee_{j\in J} tar_2^j = \mathrm{Shift}(i_{R_2}, F_1)$)	B.5	B.3
SC_1-2	$src_2' = \exists s_2 = \mathrm{L}(\mathsf{f2f}', tar_2)$	B.6	B.4
SC_1-3	$src_2 = src_2' = \exists s_2$	–	B.5
SC_1-4/5	$seq_1 = src_2 \Rightarrow_{\mathsf{f2f}'} tar_2$	B.7	B.6
SC_k-1	$tar_1 = \exists t_1$	B.9	B.7
SC_k-1$^+$	$src_2^+ = \exists(s_1' \circ s_2)$	–	B.8
SC_k-2	$src_1' = \exists s_1 = \mathrm{L}(\mathsf{f2f}', tar_1)$	B.10	B.9
SC_k-3	$src_1 = src_1' = \exists s_1$	–	B.10
SC_k-4/5	$seq_2 = src_1 \Rightarrow_{\mathsf{f2f}'} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$	B.11	B.11

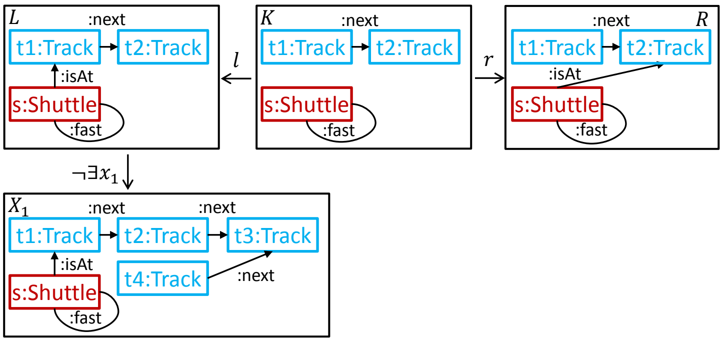


Figure 6.10. – Graph rule $\mathsf{f2f} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \neg\exists x_1, \mathrm{true}\rangle$

Example 6.18 (Seq-construction, example system with a non-trivial negative application condition)**.** In order to illustrate the consequences of a non-trivial left application condition in rules, consider the graph rule $\mathsf{f2f} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), ac_{L_2}, \mathrm{true}\rangle$ with $ac_{L_2} = \neg\exists x_1$ introduced in Example 6.1 (p. 111) and depicted again in Figure 6.10. With the additional negative application condition, the rule is only applicable if the track the shuttle is supposed to move to does not have a switch as a subsequent track. However, the target track itself may be a switch: rules check for switches two tracks ahead, not one. The composed forbidden pattern $\mathcal{F} = \neg F_1$ remains unchanged.

This case is explained in more detail in Examples B.12 and B.13 in Appendix B; here, we will only provide a short overview, with Table 6.3 listing the computation steps.

Table 6.3. – Computation steps of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$

step	computation	Figure	Example
SC_1-1	$tar_2 = \exists t_2$ (and $\bigvee_{j\in J} tar_2^j = \mathrm{Shift}(i_{R_2}, F_1)$)	B.5	B.3
SC_1-2	$src_2' = \exists s_2 = \mathrm{L}(\mathsf{f2f}, tar_2)$	B.6	B.4
SC_1-3	$src_2 = \exists(s_2, ac_{S_2}) = \exists(s_2, \mathrm{Shift}(s_2, ac_{L_2}))$	B.14	B.12
SC_1-4/5	$seq_1 = src_2 \Rightarrow_{\mathsf{f2f}} tar_2$	–	B.12
SC_k-1	$tar_1 = \exists(t_1, ac_{T_1}) = \exists(t_1, \mathrm{Shift}(s_1', ac_{S_2}))$	–	B.13
SC_k-1$^+$	$src_2^+ = \exists(s_1' \circ s_2, ac_{T_1})$	–	B.13
SC_k-2	$src_1' = \exists(s_1, ac_{S_1}') = \mathrm{L}(\mathsf{f2f}, tar_1)$	–	B.13
SC_k-3	$src_1 = \exists(s_1, ac_{S_1}) = \exists(s_1, ac_{S_1}' \wedge \mathrm{Shift}(s_1, ac_{L_1}))$	B.15	B.13
SC_k-4/5	$seq_2 = src_1 \Rightarrow_{\mathsf{f2f}'} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$	6.11	B.13

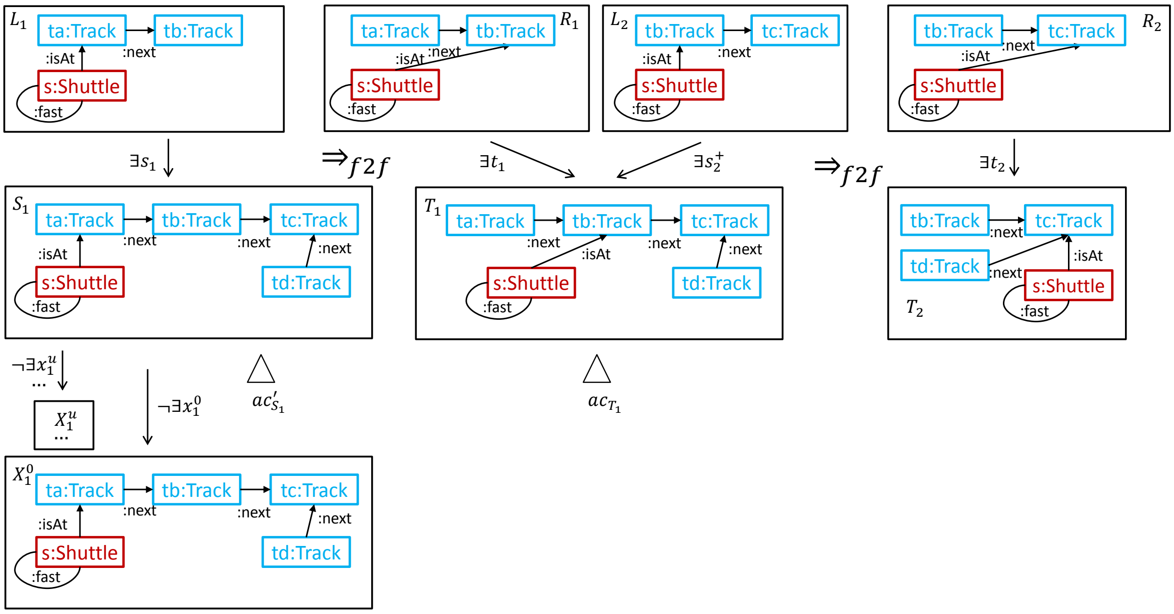


Figure 6.11. – S/t-pattern sequence $seq_2 = \exists(s_1, ac_{S_1}) \Rightarrow_{\mathsf{f2f}} (\exists(t_1, ac_{T_1}), \exists(s_2^+, ac_{T_1})) \Rightarrow_{\mathsf{f2f}} \exists t_2$ with $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$

Figure 6.11 depicts an s/t-pattern sequence seq_2 with $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$. The non-trivial negative application conditions in f2f lead to composed negative application conditions ac_{T_1} and ac_{S_1} over T_1 and S_1, respectively. In particular, $src_1 = \exists(s_1, \neg\exists x_1^0 \wedge \dots)$. We have singled out the condition fragment $\neg\exists x_1^0$ because x_1^0 is an isomorphism – and hence, no morphism can satisfy $\exists(s_1, \neg\exists s_1^0 \wedge \dots)$, which makes src_1 equivalent to false. As a result, no satisfying transformation sequence can exist for seq_2.

In particular, *trans* from Example 6.17, Figure 6.9 (p. 136) (after substituting f2f′ by f2f) is not a valid transformation sequence because of the rule's negative application condition. The interaction of two rules f2f in the fashion described by seq_2 cannot lead to the combination of right rule side and forbidden pattern used to create the rightmost target pattern in step SC$_1$-1. If this were true for all s/t-pattern sequences in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$ (using f2f), there would be no transformation sequences that could lead to a violation of the safety property. This result would not be unexpected: the rule's negative application condition was added with the intent of preventing violations of our safety property after rule application. △

In the following paragraphs, we will delve deeper into the details of the Seq-construction on a general level. Detailed examples are listed in Appendix B.

SC$_1$-1: For each rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$, $\mathrm{Shift}(i_R, C) = \bigvee_{j \in J_b} tar_{b,j}$ is a disjunction of target patterns over R of the form $tar_{b,j} = \exists(t_j, \mathrm{ac}_{T_j})$.

$$\begin{tikzcd}
ac_L \triangleright L & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[d, hook, "t_j"] & \varnothing \arrow[l, hook', "i_R"'] \arrow[d, hook, "i_P"] \\
 & & ac_{T_j} \triangleright T_j & P \triangleleft ac_P \arrow[l, hook', "\dots"']
\end{tikzcd}$$

Given a rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle$, $\mathrm{Shift}(i_R, C)$ transfers the constraint C (here: $C = \exists(i_P, ac_P)$) to the context of the right rule side R in order to create the sequences' rightmost target patterns. The result encodes satisfiability of C in the context of the result of a rule application (i.e. occurrence of the right rule side). In particular, due to the structure of graph patterns (such as C), the result will always be a disjunction of target patterns, i.e. of existential conditions over the right rule side with a nested composed negative application condition. In other words, $\mathrm{Shift}(i_R, C) = \bigvee_{j \in J_b} \exists(t_j, ac_{T_j})$ for morphisms $t_j : R \hookrightarrow T_j$. Also, note that this result will always be finite (as is any result of a Shift-construction).

The Shift-construction guarantees that, given any rule application's comatch $m' : R \hookrightarrow G$ for any graph G, m' will satisfy $\mathrm{Shift}(i_R, C)$ if and only if G satisfies the constraint C. Given the result of $\mathrm{Shift}(i_R, C)$ as explained above, we can be more specific: m' will satisfy one of the target patterns $tar_j = \exists(t_j, ac_{T_j})$ if and only if $G \vDash C$. Hence, any of the target patterns is a candidate for representing the result of a rule application that leads to satisfaction of C. Therefore, we separate the target patterns; each target pattern then spawns a new s/t-pattern sequence. While we increase the number of sequences, we reduce the complexity of individual sequences. This is different to our general approach: there, the disjunction of target patterns would just be considered one target pattern and hence, each combination of right rule side and constraints would only spawn one rather complex target pattern.

Consider the role of the construction with respect to the properties to be fulfilled by the Seq-construction: any satisfying transformation sequence implies $m' \vDash tar_j$ for the respective target pattern, which implies satisfaction of C by G. Conversely, $G \vDash C$ and the existence of the respective comatch will imply $m' \vDash tar_j$ for a specific j guaranteeing that there is a representation for the respective transformation sequence. Since the construction needs to take all rules in a rule set into account, $\mathrm{Shift}(i_R, C)$ and the splitting of the target patterns have to be computed for all rules (and right rule sides R, respectively).

Given the specific nature of graph patterns and our construction in the restricted formal model, we can provide a more intuitive view of target patterns: each target pattern is a possible overlapping of the right rule side and the graph pattern (that is, its context). In their entirety, all target patterns thusly constructed then describe all possible overlappings.

Note that a target pattern tar_j thusly created may be contradictory: if one of the negative application condition in the composed negative application condition ac_{T_j} may contain an isomorphism, making the negative application condition and the target pattern unsatisfiable. Our implementation takes care of those cases and automatically discards such target patterns. However, in our formal description, this aspect is only considered in a subsequent analysis step, not in the construction of s/t-pattern sequences.

SC$_1$-2: For each such target pattern $tar_{b,j}$, $src'_{b,j} = \mathrm{L}(b, tar_{b,j})$ is a source pattern over L of the form $src'_{b,j} = \text{false}$ or $src'_{b,j} = \exists(s_j, ac'_{S_j})$.

$$\begin{tikzcd}
ac_L \triangleright L \arrow[d, hook, "s_j"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] & R \arrow[d, hook, "t_j", "ac_{T_j} \triangleleft"'] & \varnothing \arrow[l, hook', "i_R"'] \arrow[d, hook, "i_P"] \\
ac'_{S_j} \triangleright S_j & D_j \arrow[l, hook'] \arrow[r, hook] & T_j & P \triangleleft ac_P \arrow[l, hook', "\cdots"']
\end{tikzcd}$$

After having established one or (usually) more target patterns $\mathrm{Shift}(i_R, C) = \bigvee_{j \in J_b} tar_j$ in the previous step, $src'_j = \mathrm{L}(b, tar_j)$ will transform each target pattern into a source pattern over the respective rule's left side. Intuitively speaking, given a target pattern, the rule will be applied in reverse direction to determine the (symbolic) state before rule application. In particular, by the L-construction, any rule application $G' \Rightarrow_{b,m,m'} G$ will imply the equivalence of $m \vDash src'$ and $m' \vDash tar$: if the source pattern is satisfied before rule application, the target pattern will be satisfied after rule application and the resulting graph will satisfy C (see step SC$_1$-1).

Again, the result of this construction is finite. Given a number of target patterns procuded by the previous step, the result of this step is an equal number of corresponding source patterns.

SC$_1$-3: For the latter case, $src_{b,j} = \exists(s_j, ac_{S_j})$ with $ac_{S_j} = ac'_{S_j} \wedge \text{Shift}(s_j, ac_L \wedge \text{Appl}(b))$ is a source pattern.

This step is concerned with the applicability of the rule in question for any transformation sequence satisfying the s/t-pattern sequence under construction. Given a source pattern $src' = \exists(s, ac'_S)$ with $s : L \hookrightarrow S$, the respective left application condition ac_L and the applicability condition $\text{Appl}(b)$ is transferred to the context of the source pattern's graph – S – by shifting it over the morphism s. Then, it is conjunctively joined to the nested composed negative application condition ac'_S, thusly creating the new source pattern $src = \exists(s, ac_S)$ with $ac_S = ac'_S \wedge \text{Shift}(s, ac_L \wedge \text{Appl}(b))$. Any potential match $m : L \hookrightarrow G'$ to any graph G' that satisfies src will then imply satisfaction of $ac_L \wedge \text{Appl}(b)$ by m, which is required for the existence of a transformation $G' \Rightarrow_{b,m,m'} G$ (by Lemma 2.30 (p. 37)).

Note that this step is slightly more involved than its counterpart in the general approach, where the left application condition and applicability condition are conjunctively joined to the existing source pattern. Here, that procedure would result in an application condition that does not fit our restricted formal model because it does not conform to our definition of source patterns. Hence, it is necessary to shift the condition over the source pattern's existential condition's morphism and conjunctively combine it with the existing condition ac'_S.

Often, this step will leave the source pattern effectively unchanged: shifting a trivial left application conditions $ac_L = \text{true}$ does not change the source pattern. Furthermore, rules that do not delete nodes have a trivial applicability condition $\text{Appl}(b) = \text{true}$. If both cases apply, this step will result in $src_{b,j} = src'_{b,j}$ (from the previous step).

It is important to highlight that the left application condition and applicability condition is only transferred to the context of the source pattern; the target pattern remains unchanged. While the information about the shifted application condition will be considered in further computation steps that prolong the sequence to the left, is is not propagated to the right (in this case, the target pattern). For the general approach, this was not an issue because the analysis of s/t-pattern sequences only needed to consider the leftmost source pattern, where all accumulated information is available. This will not be the case for our restricted formal model. In Chapter 7, Section 7.1, we will discuss the notion of *forward propagation*, which addresses this issue.

SC$_1$-4: For each such pair of source and target pattern $src_{b,j}$ and $tar_{b,j}$, $src_{b,j} \Rightarrow_b tar_{b,j}$ is a 1-sequence of s/t-patterns.

SC$_1$-5: Finally, we define $\text{Seq}^r_1(\mathcal{R}, C) = \{src_{b,j} \Rightarrow_b tar_{b,j} \mid b \in \mathcal{R} \wedge j \in J_b\}$ as the set of these sequences.

These steps put the computed target patterns and corresponding source patterns together to form a number of 1-sequences of source/target patterns. Per rule, that number is equal to the number of operands in the disjunction that is the result of $\text{Shift}(i_R, C)$ in step SC$_1$-1 – minus the number of cases where a source pattern has been computed as $src = \text{L}(b, tar) = \text{false}$ in step SC$_1$-2.

By Theorem T.1r (p. 130), any transformation sequence $trans = G_0 \Rightarrow_{\mathcal{R}} G_1$ that leads to C has a representing s/t-pattern sequence seq in $\text{Seq}^r_1(\mathcal{R}, C)$, i.e. $trans \models seq$. Also, given a transformation sequence $trans$ that satisfies an s/t-pattern sequence $seq \in \text{Seq}^r_1(\mathcal{R}, C)$, we know that $trans$ leads to C.

SC$_\mathsf{k}$-1: For each sequence $seq = src_1 \Rightarrow_{b_1} ... \Rightarrow_{b_k} tar_k \in \mathrm{Seq}^r_k(\mathcal{R}, C)$ with $src_1 = \exists(s_1 : L_1 \hookrightarrow S_1, ac_{S_1})$, each $b = \langle(L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$, and each graph T_j and pair of injective and jointly surjective morphisms $(t_j : R \hookrightarrow T_j, s'_j : S_1 \hookrightarrow T_j)$, $tar_{b,j} = \exists(t_j, ac_{T_j})$ with $ac_{T_j} = \mathrm{Shift}(s'_j, ac_{S_1})$ is a target pattern over R.

$$\begin{tikzcd}
ac_L \triangleright L & K \arrow[l, hook, "l"'] \arrow[r, hook, "r"] & R \arrow[d, hook, "t_j"] & L_1 \arrow[d, hook, "s_1"] & \varnothing \arrow[l, hook, "i_{L_1}"] \arrow[ll, hook, bend right, "i_R"'] \arrow[dl, hook, "i_{S_1}"] \\
& & ac_{T_j} \triangleright T_j & S_1 \triangleleft ac_{S_1} \arrow[l, hook, "s'_j"'] &
\end{tikzcd}$$

The following steps SC$_\mathrm{k}$-1 to SC$_\mathrm{k}$-5 take all constructed s/t-pattern sequences of length k and prolongs them to create the required sequences of length $k + 1$. Intuitively, step SC$_\mathrm{k}$-1 applies the same idea as step SC$_1$-1, but considers the leftmost source patterns of the sequences of length k instead of the constraint C. In other words, the target patterns resulting from this step encode all situations where a rule application leads to the start of a s/t-pattern sequence of length k that fulfills the required properties with respect to the constraint C – i.e., its satisfying transformation sequences lead to C.

Finding all injective and jointly surjective morphism pairs (t_j, s'_j) corresponds to computing $\mathrm{Shift}(i_R, src_{1|\varnothing}) = \mathrm{Shift}(i_R, \exists(s_1 \circ i_{L_1}, ac_{S_1})$. In fact, $\mathrm{Shift}(i_R, \exists(s_1 \circ i_{L_1}, ac_{S_1}))$ equals the disjunction $\bigvee_{j \in J} \exists(t_j, \mathrm{Shift}(s'_j, ac_{S_1}))$. We create a new target pattern (and, in step SC$_\mathrm{k}$-4/5, a new s/t-pattern sequence) for each morphism pair (t_j, s') – in particular, $tar_j = \exists(t_j, \mathrm{Shift}(s'_j, ac_{S_1}))$. This is similar to step SC$_1$-1, where we split the disjunction of target patterns into several individual target patterns.

This procedure is different to its counterpart in the general approach: there, we considered morphism pairs with the right and the left rule sides as domains. Here, we use the right rule side and the source pattern's context graph S_1. Since a source pattern in our restricted formal model embeds a left side into the additional context provided by the codomain of the source pattern's existential condition's morphism, it makes sense to use that context. In other words, the resulting target patterns each describe specific combinations – overlappings – between the right rule side and the source pattern's context, similar to target patterns from step SC$_1$-1 describing all overlappings between the right rule side and the graph pattern C.

SC$_\mathsf{k}$-1$^+$: For each such target pattern $tar_{b,j}$, $src^+_{1,j} = \exists(s'_j \circ s_1, ac_{T_j})$ is a source pattern over L_1 and $(tar_{b,j}, src^+_{1,j})$ is a target/source pattern over (b, b_1).

By the definition of s/t-pattern sequences (Definition D.2 (p. 82)), a s/t-pattern sequence of length 2 (or longer) is not merely a list of two (or more) 1-sequences. Rather, they have to be glued together by combining a target and a subsequent source pattern to form a target/source pattern. In particular, the target patterns created in the previous steps need to be connected to the s/t-pattern sequence used in its creation by creating target/source patterns. Without this connection, it would not be clear how the result of the previous rule application and the match of the subsequent rule application interact.

By construction, each newly created target pattern has the form $tar_{b,j} = \exists(t_j, ac_{T_j})$, where $t_j : R \hookrightarrow T_j$ and $s'_j : S_1 \hookrightarrow T_j$ form an injective and jointly surjective morphism pair; furthermore, we have a source pattern $src_1 = \exists(s_1, ac_{S_1})$. By construction, we have $ac_{T_j} = \mathrm{Shift}(s'_j, ac_{S_1})$. Since T_j – for the respective target pattern – describes the context more precisely than S_1 in the source pattern src_1, we can extend src_1 to include the additional context provided (via the right rule side R) in T_j. Then, for the target pattern $tar_{b,j}$, we have an extended source pattern

$src^+_{j,1} = \exists(s'_j \circ s_1, ac_T)$. For each target pattern $tar_{b,j}$ created in step SC$_\mathrm{k}$-1, this step will create a different source pattern $src^+_{1,j}$ because the morphism s'_j will differ between morphism pairs.

With respect to notation, note that src^+ usually denotes the extension of a source pattern src to a source pattern src^+ that is used as part of a target/source pattern. Reversely, src^- refers to an 'underlying' source pattern, that has been extended to a source pattern src appearing in a target/source pattern.

SC$_\mathbf{k}$-2: For each such target pattern $tar_{b,j}$, $src'_{b,j} = \mathrm{L}(b, tar_{b,j})$ is a source pattern over L of the form $src'_{b,j} = \text{false}$ or $src'_{b,j} = \exists(s_j, ac'_{S_j})$.

$$\begin{tikzcd}
ac_L \triangleright L \arrow[d, hook, "s_j"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] & R \arrow[d, hook, "t_j", "ac_{T_j} \triangleleft"'] & L_1 \arrow[d, hook, "s_1"] \\
ac'_{S_j} \triangleright S_j & D_j \arrow[l, hook'] \arrow[r, hook] & T_j & S_1 \triangleleft ac_{S_1} \arrow[l, hook', "s'_j"']
\end{tikzcd}$$

As in step SC$_1$-2, this step transfers all newly constructed target patterns over the rule in reverse direction to create corresponding source pattern over the respective rules' left sides.

SC$_\mathbf{k}$-3: For the latter case, $src_{b,j} = \exists(s_j, ac_{S_j})$ with $ac_{S_j} = ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b))$ is a source pattern over L.

Similar to step SC$_1$-3, we shift the rule's left application condition and applicabiliy condition to the context of the source pattern, unless that source pattern – computed in the previous step – has the logical value false. Again, the purpose is to have a satisfying match also satisfy the rule's left application condition and applicability condition so that the rule can be applied.

SC$_\mathbf{k}$-4: For each such pair of source and target pattern $src_{b,j}$ and $tar_{b,j}$, $src_{b,j} \Rightarrow_b tar_{b,j}, src^+_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$ is a k+1-sequence of s/t-patterns.

SC$_\mathbf{k}$-5: Finally, we define $\mathrm{Seq}^r_{k+1}(\mathcal{R}, C) = \{src_{b,j} \Rightarrow_b tar_{b,j}, src^+_{1,j} \Rightarrow \ldots \Rightarrow tar_k \mid b \in \mathcal{R} \wedge j \in J_b \wedge seq \in \mathrm{Seq}^r_k(\mathcal{R}, C)\}$ as the set of these sequences.

In these steps, we combine the newly created and corresponding target/source patterns and source patterns in order to create all s/t-pattern sequences of length $k+1$. The target/source patterns (step SC$_\mathrm{k}$-1$^+$) connect the created target patterns (step SC$_\mathrm{k}$-1) to the leftmost source patterns computed in the previous iteration (i.e. for length k) of the Seq-contruction. The result is a finite set of s/t-pattern sequences of length $k+1$. From here on, steps SC$_\mathrm{k}$-1 to SC$_\mathrm{k}$-5 can be repeated until the desired length of the s/t-pattern sequences is reached.

By Theorem T.1r, any transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \ldots \Rightarrow_{b_k,m_k,m'_k} G_k$ that leads to C has a representing s/t-pattern sequence seq in $\mathrm{Seq}^r_k(\mathcal{R}, C)$, i.e. $trans \models seq$. Also, given a transformation sequence $trans$ that satisfies an s/t-pattern sequence $seq \in \mathrm{Seq}^r_k(\mathcal{R}, C)$, we know that $trans$ leads to C.

With the Seq-construction established, we can move on to the analysis, with different approaches depending on whether we want to verify a k-inductive invariant, perform k−1-bounded backward model checking, or verify an operational invariant by combining the former and the latter case.

6.4. k-Inductive Invariant Checking

As established, verifying a k-inductive invariant under a guaranteed constraint amounts to finding all transformation sequences of length k that lead to a violation of the invariant and analyzing the graphs in the sequence for violations of the invariant or the guaranteed constraint. For our restricted formal model, Lemma 6.10 (p. 123) further refines that process: given a composed forbidden pattern $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$, we consider all transformation sequences of length k leading to violations of $\mathcal{F}$ by satisfying one of the individual forbidden patterns F_i. Then, we analyze those sequences for violations of the composed guaranteed pattern $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$, i.e. for occurences of H_j, and for violations of the composed forbidden pattern earlier in the sequence, i.e. for occurences of F_i in all graphs except the last one. Of course, we do not analyze the actual transformation sequences, but their symbolic representations – the s/t-pattern sequences constructed by $\mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F}) = \bigcup_{i \in I} \mathrm{Seq}_k^r(\mathcal{R}, F_i)$. This process is described by our central theorem for verifying k-inductive invariants in the restricted formal model:

Theorem T.2r (k-inductive invariant checking [3]). *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ and $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$ be a composed forbidden pattern and composed guaranteed pattern, respectively.*

$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ if, for all sequences $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ with $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$, one of the following conditions holds:

1. $\exists z, v(1 \leq z \leq k \wedge (src_{z|\varnothing} \vDash H_v \vee src_{z|\varnothing} \vDash F_v))$.
2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.

Proof. According to Lemma 6.10 (p. 123), we need to show that for all k-sequences of transformations $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$, it holds that:

$$\exists u(G_k \vDash F_u) \Rightarrow \exists z, v(0 \leq z \leq k \wedge G_z \vDash H_v) \vee \exists z, v(0 \leq z \leq k-1 \wedge G_z \vDash F_v)$$

Consider an arbitrary k-sequence of transformations to $\mathcal{R}$ (with corresponding graphs) $trans = G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ such that $\exists u(G_k \vDash F_u)$ with, for ease of reading, $F_u = F$. More specifically, $trans = G_0 \Rightarrow_{b_1, m_1, m'_1} \dots \Rightarrow_{b_k, m_k, m'_k} G_k$ for rules $b_i \in \mathcal{R}$ and matches (comatches) m_i (m'_i) and $trans$ leads to F. We want to show that $G_z \vDash H_v$ for $0 \leq z \leq k$ and $v \in J$ or that $G_z \vDash F_v$ for $0 \leq z \leq k-1$ and $v \in I$.

By Theorem T.1r (p. 130), there is a k-sequence of s/t-patterns $seq \in \mathrm{Seq}_k^r(\mathcal{R}, F)$ (and hence, $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$) with $trans \vDash seq$. By precondition, one of the following is true:

1. There exist z, v with $1 \leq z \leq k$ such that $src_{z|\varnothing} \vDash H_v$ or $src_{z|\varnothing} \vDash F_v$. Because of $trans \vDash seq$, we have $m_z \vDash src_z$ and, with $m_z : L_z \hookrightarrow G_{z-1}$ and Lemma 2.38 (p. 43), we gain $G_{z-1} \vDash src_{z|\varnothing}$ and, by implication of graph constraints (Definition 2.36 (p. 42)), $G_{z-1} \vDash H_v$ or $G_{z-1} \vDash F_v$.
2. There exists v such that $tar_{k|\varnothing} \vDash H_v$. Because of $trans \vDash seq$, we have $m'_k \vDash tar_k$ and, with $m'_k : R_k \hookrightarrow G_k$ and Lemma 2.38 (p. 43), we gain $G_k \vDash tar_{k|\varnothing}$ and, by implication of graph constraints (Definition 2.36 (p. 42)), $G_k \vDash H_v$.

Hence, $\mathcal{F}$ is a k-inductive invariant for GTS under H. □

Note that, for an s/t-pattern sequence's intermediate target/source patterns, it does not make a difference whether we reduce the source or target pattern to a regular pattern – the result will always be identical. However, the situation is different for the leftmost source pattern and rightmost target pattern because they do not appear as part of a target/source pattern. This is the reason for analyzing only (reduced) source patterns in the theorem's first condition and only the rightmost target pattern in the second condition.

By Lemma 6.10 (p. 123), we need to check all reduced patterns for implication of a guaranteed pattern (conditions (1) and (2)). For forbidden patterns (condition (1)), we do not consider the rightmost target pattern; obviously, it will always imply the pattern used in the Seq-construction to create the s/t-pattern sequence. Hence, the second condition only compares $tar_{k|\varnothing}$ with guaranteed patterns.

The main differences to the counterpart of Theorem T.2r in the general approach – Theorem T.2g (p. 96) – are

- the splitting of a composed forbidden pattern into individual forbidden patterns: the set $\mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$ is not computed directly, but is the union of applications of $\mathrm{Seq}_k^r(\mathcal{R}, F_i)$ for each forbidden pattern F_i,
- the analysis of intermediate source patterns and the rightmost target pattern instead of finding a satisfying transformation sequence by analyzing the leftmost source pattern, and
- the fact that Theorem T.2r (p. 143) specifies a sufficient condition and establishes an implication, not an equivalence – if the condition is not satisfied, the result is inconclusive.

Investigating violations of individual forbidden patterns separately has the aim of creating more specific counterexamples and reducing their complexity. As a result, however, the number of (symbolic) counterexamples may be higher than with our general approach.

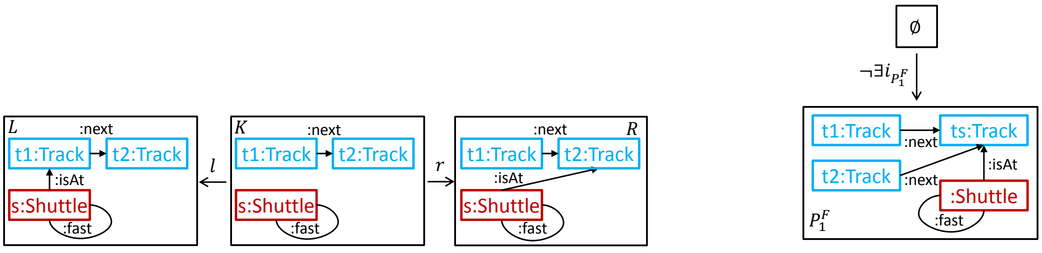
The analysis of intermediate source patterns is necessary because the intention of our restricted approach was to avoid the explosion of computational effort when accumulating all information in the Seq-construction. In a way, we replace the expensive computation of $\mathrm{Shift}(i_L, \mathcal{H})$ and $\mathrm{Shift}(i_L, \mathcal{F})$ for left rule sides in the Seq-construction of our general approach by comparing reduced source (target) patterns with individual forbidden and guaranteed patterns. In particular, that comparison is implemented in the conditions $\exists z, v(1 \le z \le k \wedge (src_{z|\varnothing} \vDash H_v \vee src_{z|\varnothing} \vDash F_v))$ and $\exists v(tar_{k|\varnothing} \vDash H_v)$, respectively.

However, an unvoidable consequence is the third difference mentioned above: Theorem T.2r establishes a sufficient condition only. Failure to discard s/t-pattern sequences as counterexamples by the analysis described in that theorem does not guarantee the invalidity of the composed forbidden pattern as a k-inductive invariant. In other words, we may have spurious counterexamples (false negatives): s/t-pattern sequences without satisfying transformation sequences that fulfil the criterion of violating the k-inductive invariant in the sense of Definition D.1 (p. 62). We will discuss their implications in Section 6.8.

This last point also means that equivalence of results between the general and restricted approach as seen in the previous examples is not always guaranteed. The restricted approach will necessarily concur with any result by the general approach. However, false negatives established by the restricted approach could be properly discarded by the general approach.

Example 6.19 (2-inductive invariant checking for an unsafe sytem)**.** This example follows and is based on Example 6.17 (p. 135) (Seq-construction) and its detailed variants in Examples B.2–B.11. It uses elements of Example 6.1 (p. 111). It is similar to Example 5.16 (p. 97) in our general approach. Again, we have a rule set $\mathcal{R} = \{\mathsf{f2f'}\}$ (Figure 6.12(a)) in a graph transformation system $GTS = (TG, \mathcal{R})$ and a composed forbidden pattern $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$ (Figure 6.12(b)). For simplicity, we will only consider a trivial guaranteed constraint $\mathcal{H} = \mathrm{true}$.

Consider the s/t-pattern sequence $seq \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$, which was also shown as seq_2 in the earlier examples and is depicted again in Figure 6.13. By Theorem T.1r (p. 130), seq represents an infinite number of transformation sequences leading to F_1, i.e. a violation of the intended 2-inductive invariant. By Theorem T.2r (p. 143), we have to find source patterns that, when reduced to a pattern, imply F; then, seq would not be a counterexample to $\mathcal{F} = \neg F_1$ being an 2-inductive invariant for GTS.



(a) Graph rule f2f′

(b) $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

Figure 6.12. – Graph rule and intended 2-inductive invariant

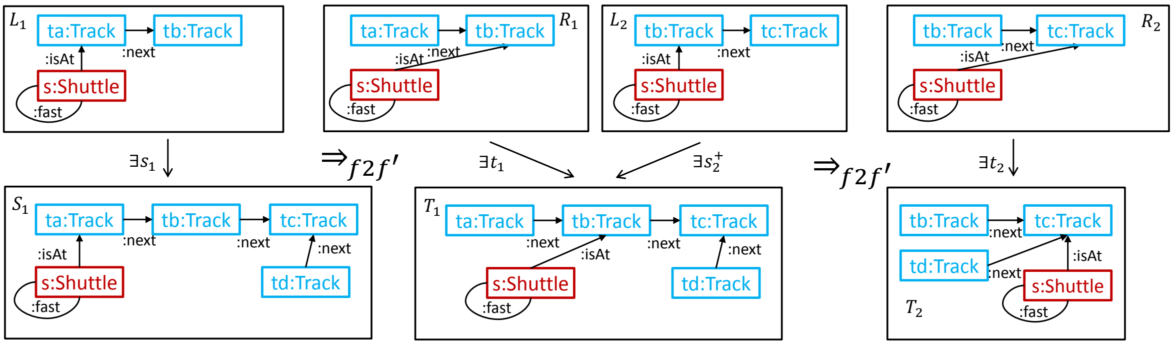
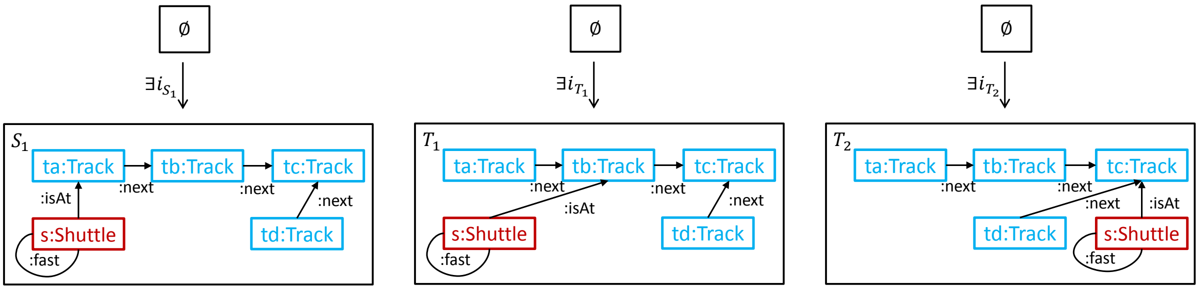


Figure 6.13. – s/t-pattern sequence $seq \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$



(a) $src_{1|\varnothing} = \exists i_{S_1}$

(b) $src_{2|\varnothing} = \exists i_{T_1}$

(c) $tar_{2|\varnothing} = \exists i_{T_2}$

Figure 6.14. – Reduced source and target patterns $src_{1|\varnothing}$, $src_{2|\varnothing}$, and $tar_{2|\varnothing}$

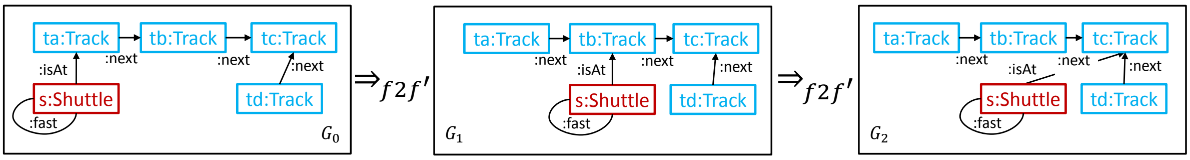


Figure 6.15. – Transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f'},m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f'},m_2,m_2'} G_2$; $trans \models seq$

In order to find out whether $src_{1|\varnothing}$ or $src_{2|\varnothing}$ imply F_1, we can apply Theorem 6.8 (p. 120). Here, its application is not computationally challenging; in fact, we can guess the result by looking at the patterns. The reduced source patterns $src_{1|\varnothing}$ and $src_{2|\varnothing}$ are depicted in Figures 6.14(a) and 6.14(b). There does not exist an injective morphism from P_1^F to S_1, T_1, or T_2. Informally, we cannot find a fast shuttle on a switch in one of the situations described by the (reduced) patterns. Hence, we cannot conclude that $src_{1|\varnothing}$ or $src_{2|\varnothing}$ imply F_1. In general we cannot be sure that $src_{1|\varnothing}$ or $src_{2|\varnothing}$ do not imply F_1: Theorem 6.8 (p. 120) establishes a sufficient, not a necessary condition. This is one reason why Theorem T.2r (p. 143) only describes a sufficient condition, too. In this case, however, the absence of non-trivial nested composed negative application conditions allows the conclusion that neither of the reduced source patterns imply F_1. For the verification approach to be sound, we need to keep counterexamples even if we cannot draw this conclusion: as long as we cannot safely discard all counterexamples, we have to assume that the composed forbidden patterns is not an inductive invariant.

By Theorem T.2r (p. 143), *seq* is a symbolic counterexample for $\mathcal{F} = \neg F_1$ being a 2-inductive invariant. Fortunately, *seq* already hints at the problem: a shuttle is allowed to continue its movement in speed mode fast even when approaching a switch. That inevitably leads to a violation of our safety property. This insight is an important difference to the result of our general approach (Example 5.16 (p. 97)); there, it is much more difficult to discern the problem by looking at a (symbolic) counterexample.

From Example 6.17 (p. 135), we also know that there is indeed a satisfying transformation sequence *trans*, which is depicted again in Figure 6.15. However, finding a satisfying transformation sequence is not usually covered in our analysis – not only is it an undecidable problem in general, but also computationally expensive. Risking the occurrence of unsatisfiable s/t-pattern sequences as counterexamples (false negatives) is part of the compromise between expressive power, termination, and computational effort.

Note that there are other s/t-pattern sequences besides *seq* in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$. Some can be discarded by the analysis, others will be retained as counterexamples. Since $\mathcal{F}$ is not a 2-inductive invariant even for a single graph rule, verifying it for the complete rule set of our running example – $\mathcal{R}' = \{\mathsf{f2f'}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{s2s}, \mathsf{s2a'}, \mathsf{a2b}, \mathsf{a2f'}\}$ – will yield a similar result. Even when taking the running example's non-trivial guaranteed constraint $\mathcal{H} = \bigwedge_{1 \le j \le 15} \neg H_j$ into account for either case, several counterexamples (including *seq*) will remain. Analysis of *seq* with respect to $\mathcal{H}$ would require considering $tar_{2|\varnothing}$ (Figure 6.14(c)) in addition to $src_{1|\varnothing}$ and $src_{2|\varnothing}$.

These results are expected: we have established in Example 5.16 (p. 97) with the general approach that $\mathcal{F}$ is not a 2-inductive invariant of GTS' under $\mathcal{H}$. Necessarily, our restricted approach will come to the same conclusion. △

Example 6.20 (2-inductive invariant checking for a safe system)**.** Now, consider a graph transformation system $GTS = (TG, \mathcal{R})$ with $\mathcal{R} = \{\mathsf{f2f}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{s2s}, \mathsf{s2a}, \mathsf{a2b}, \mathsf{a2f}\}$ (Example 6.1 (p. 111)) and (again) the composed forbidden pattern $\mathcal{F} = \neg F_1 = \neg \exists i_{P_1^F}$. The rules f2f, a2f, and s2a are shown again in Figures 6.16(b), 6.16(a), and 6.16(c), respectively; F_1 is shown again in

Figure 6.16(d). The rules shown have non-trivial composed negative application conditions in comparison to their (unsafe) counterparts a2f′, s2a′, and f2f′. Intuitively, they prevent acceleration or high speed modes – or rather, the application of the respective rules – when a switch is two tracks ahead (or, for s2a, one track ahead). We will also consider the non-trivial composed guaranteed pattern $\mathcal{H} = \bigwedge_{1 \leq i \leq 15} \neg H_i$ (Example 6.1 (p. 111)); H_1, H_5, and H_6 are depicted again in Figures 6.16(e), 6.16(f), and 6.16(g). This example is equivalent to Example 5.17 (p. 98) discussed in the general approach.

With respect to the rule f2f only, this example follows the (Seq-)constructions explained in Example 6.18 (p. 136) and, in more detail, in Examples B.12 and B.13. For the complete rule set $\mathcal{R}$, we also get a set of s/t-pattern sequences $\mathrm{Seq}^r_2(\mathcal{R}, F_1)$. This set includes

$$seq = \exists(s_1, \neg\exists x_1^0 \wedge \bigwedge_{u \in U} \neg\exists x_1^u \wedge ac'_{S_1}) \Rightarrow_{\mathsf{f2f}} (\exists(t_1, ac_{T_1}), \exists(s_2^+, ac_{T_1})) \Rightarrow_{\mathsf{f2f}} \exists t_2,$$

which is depicted in Figure 6.16(h). However, *seq* can be discarded by application of Theorem T.2r (p. 143). In particular, the source pattern $\exists(s_1, \neg\exists x_1^0 \wedge \bigwedge_{u \in U} \neg\exists x_1^u \wedge ac'_{S_1})$ has a composed negative application condition that contains a negative application condition $\neg\exists x_1^0$ with an isomorphism; then, src_1 is logically equivalent to false. As a result, so is $src_{1|\varnothing}$, which then implies all possible graph constraints – in particular, it implies F_1. Hence, *seq* can be discarded as a counterexample. This is expected: the condition $\neg\exists x_1^0$ in the source pattern was the result of taking the composed negative application conditions of f2f into account (during step $\mathrm{SC_k}$-3 of the Seq-construction). Indeed, in this case, the conditions prevent a violation of the intended inductive invariant.

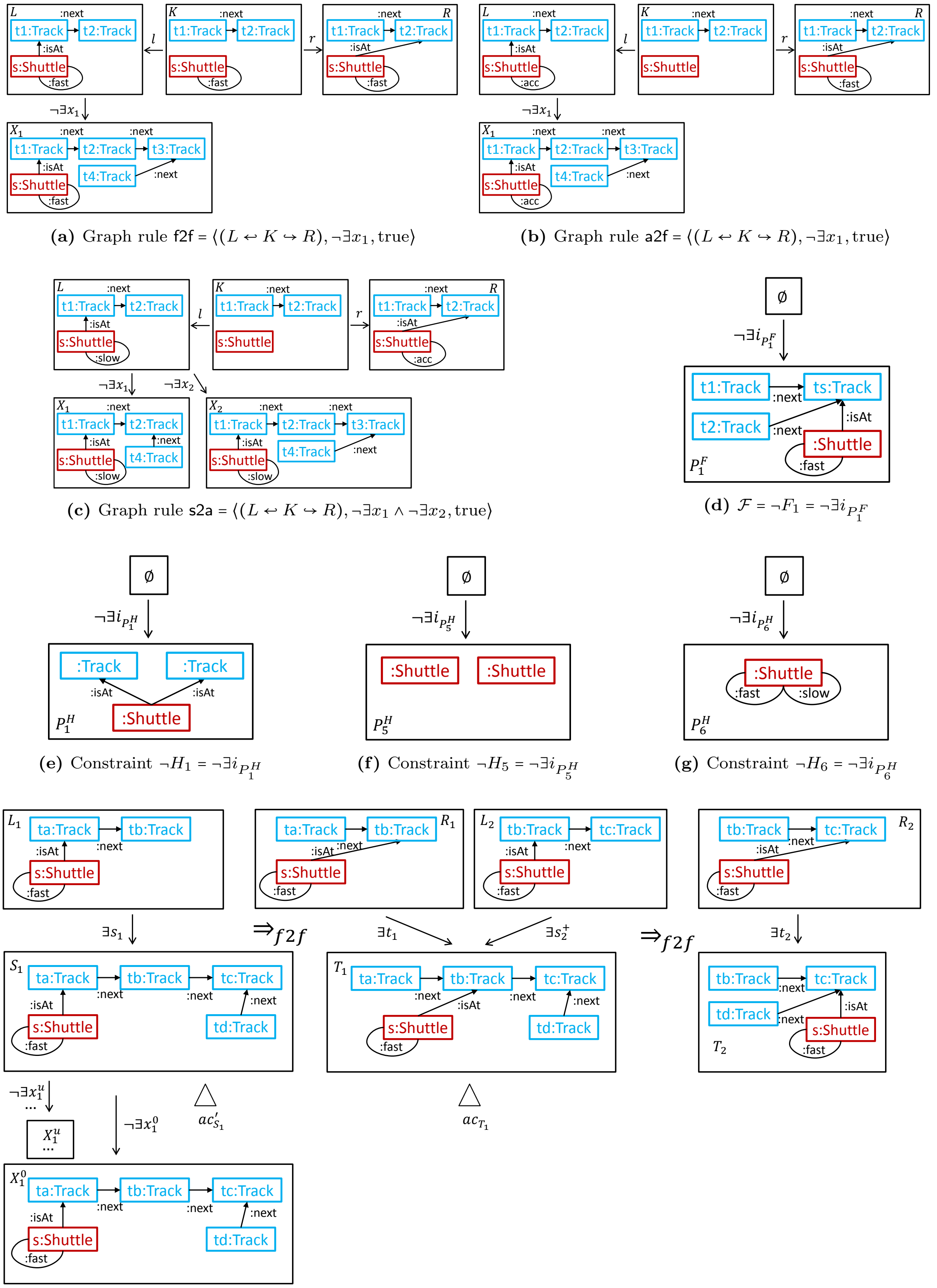
There are other s/t-pattern sequences in $\mathrm{Seq}^r_2(\mathcal{R}, F_1)$; for instance, consider $tar'_2 = \exists t_2$ depicted in Figure 6.17. This target pattern is one of the results of step $SC_1{-}1$ in the Seq-construction $\mathrm{Seq}^r_2(\mathcal{R}, F)$ for the rule f2f and will, with subsequent application of the remaining steps, spawn a number of s/t-pattern sequences to be analyzed. However, all of those s/t-pattern sequences will be discarded: T'_2 contains both P_1^H (of the guaranteed pattern $H_1 = \exists i_{P_1^H}$, Figure 6.16(e)) and P_5^H (of the guaranteed pattern $H_5 = \exists i_{P_1^H}$, Figure 6.16(f)) as a subgraph. Hence, by Theorem 6.8 (p. 120), $tar'_{2|\varnothing}$ implies H_1 and H_5 and, by Theorem T.2r (p. 143), all sequences containing tar'_2 are discarded – any satisfying transformation sequences would violate both guaranteed patterns. Note that this particular target pattern will also have been created during the verification of Example 6.19 (p. 144), although it is not explicitly mentioned there.

Formally, the respective s/t-pattern sequences will only be discarded after all sequences have been computed and Theorem T.2r is applied to all sequences. In implementation, however, it makes sense to stop continuing the computation of s/t-pattern sequences with tar'_2 as the rightmost target pattern right after that target pattern has been created. Since T'_2 can not 'lose' any elements during the subsequent computation steps, P_1^H and P_5^H will always be contained in the target pattern's context and the corresponding guaranteed patterns will always be violated.

Similar to Example 5.17 (p. 98), we can extend our composed forbidden pattern such it also forbids accelerating or braking shuttles on a switch. The corresponding graph constraints $\neg F_2$ and $\neg F_3$ from Example 6.1 (p. 111) are shown again in Figure 6.18. Then, our new composed forbidden pattern is $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ – and $\mathcal{F}$ is a 2-inductive invariant for GTS under $\mathcal{H}$.

As discussed before, the value for 2 is indirectly encoded in the rules' negative application conditions and the speed mode protocol. Furthermore, with respect to $\mathcal{F}$ being a 2-inductive invariant under $\mathcal{H}$, we get the same results as in the general approach. △

This concludes the restricted approach to k-inductive invariant checking. With inductive invariant checking as the inductive step of our inductive verification approach, we will discuss its base case next: $k{-}1$-bounded backward model checking (for the restricted formal model).



(a) Graph rule f2f = $\langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1, \text{true}\rangle$

(b) Graph rule a2f = $\langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1, \text{true}\rangle$

(c) Graph rule s2a = $\langle (L \hookleftarrow K \hookrightarrow R), \neg\exists x_1 \wedge \neg\exists x_2, \text{true}\rangle$

(d) $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

(e) Constraint $\neg H_1 = \neg\exists i_{P_1^H}$

(f) Constraint $\neg H_5 = \neg\exists i_{P_5^H}$

(g) Constraint $\neg H_6 = \neg\exists i_{P_6^H}$

(h) $seq = src_1 \Rightarrow_{\mathsf{f2f}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$ with $seq \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$

Figure 6.16. – Fragments of an example system with a 2-inductive invariant $\mathcal{F} = \neg\mathcal{F}_1$

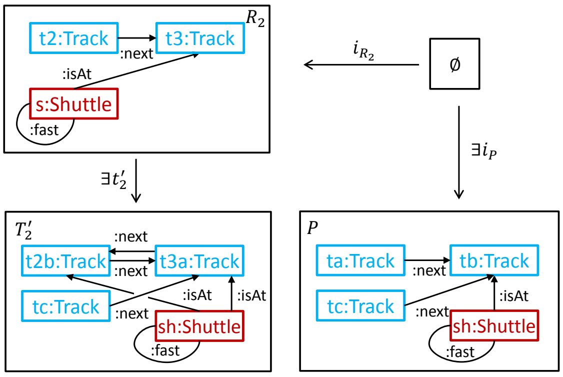
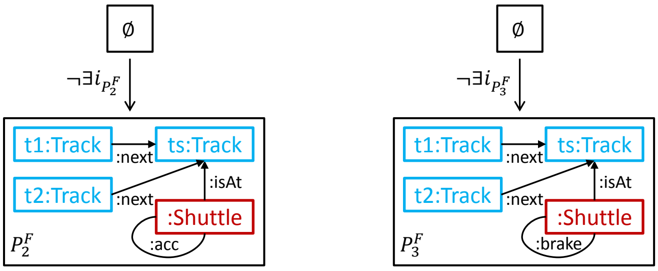


Figure 6.17. – Target pattern $tar'_2 = \exists t'_2$, created in step SC$_1$-1 of Seq$^r_2(\mathcal{R}, F_1)$ for rule f2f



(a) $\neg F_2 = \neg\exists i_{P^F_2}$

(b) $\neg F_3 = \neg\exists i_{P^F_3}$

Figure 6.18. – Extended safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$

6.5. k−1-Bounded Backward Model Checking

By Lemma 6.11 (p. 124), verifying the validity of a composed forbidden pattern $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ in the k−1-bounded state space under a composed guaranteed pattern $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$ for induced graph grammars IND$(GTS, \mathcal{S})$ amounts to the following steps: finding all transformation sequences of length smaller than k that lead to one of the forbidden patterns F_i; then, analyzing each sequence for violations of guaranteed patterns in one graph or violations of start configuration patterns in the leftmost graph. The former case allows us to discard sequences because of the restriction of the state space by the guaranteed constraint $\mathcal{H}$ (cf. Definition 4.1 (p. 55)); the latter case means that the sequence, while leading to a violation of $\mathcal{F}$, does not originate in a possible start configuration.

For our approach, we use s/t-pattern sequences as the symbolic representation of transformation sequences leading to $\neg\mathcal{F}$. We compute SEQ$^r_{k-1}(\mathcal{R}, \neg\mathcal{F}) = \bigcup_{1 \le i \le k-1}$ Seq$^r_i(\mathcal{R}, \neg\mathcal{F})$ and perform analysis of s/t-pattern sequences with respect to guaranteed constraints and start configuration constraints similar to Theorem T.2r (p. 143). A proper symbolic counterexample is an s/t-pattern sequence of length smaller than k that has satisfying transformation sequences leading to $\mathcal{F}$ under $\mathcal{H}$ and originating in a graph that also satisfies the composed start configuration pattern $\mathcal{S}$. On the other hand, we can discard s/t-pattern sequences whose satisfying transformation sequences violate $\mathcal{H}$ at some point or do not start with a graph satisfying $\mathcal{S}$.

Theorem T.3r (k−1-bounded backward model checking)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o \in O} \neg SC_o$ be a composed forbidden pattern, composed guaranteed pattern, and composed start configuration pattern, respectively, with $\mathcal{S} \models \mathcal{F}$.*

For all graphs $G \in$ REACH$_{k-1}(GG, \mathcal{H})$ and graph grammars $GG = (GTS, G_0)$ with $GG \in$ IND$(GTS, \mathcal{S})$, we have $G \models \mathcal{F}$, if for all sequences $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_n} tar_n$ with $seq \in$

$\mathrm{SEQ}^r_{k-1}(\mathcal{R},\neg\mathcal{F})$, *one of the following conditions holds:*

1. $\exists z, v(1 \leq z \leq n \wedge (src_{z|\varnothing} \vDash H_v))$.
2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.
3. $\exists v(src_{1|\varnothing} \vDash SC_v)$.

Proof. According to Lemma 6.11 (p. 124), we need to show that for all sequences of transformations $G_0 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n$ with $0 \leq n \leq k$, it holds that

$$\exists u(G_n \vDash F_u) \Rightarrow (\exists z, v(0 \leq z \leq n \wedge G_z \vDash H_v) \vee \exists v(G_0 \vDash SC_v))$$

Consider an arbitrary n-sequence of transformations to $\mathcal{R}$ (with corresponding graphs) $trans = G_0 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n$ such that $0 \leq n \leq k$ and $\exists u(G_n \vDash F_u)$ with, for ease of reading, $F_u = F$. More specifically, $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \ldots \Rightarrow_{b_n,m_n,m'_n} G_n$ for rules $b_i \in \mathcal{R}$ and matches (comatches) m_i (m'_i) and $trans$ leads to F. We want to show that there exists a z with $0 \leq z \leq n$ such that $G_z \vDash H_v$ for $v \in J$ or that $G_k \vDash SC_v$ for $v \in O$.

By Theorem T.1r, there is an n-sequence of s/t-patterns $seq \in \mathrm{Seq}^r_n(\mathcal{R}, F)$ with $trans \vDash seq$. Then, $seq \in \mathrm{SEQ}^r_{k-1}(\mathcal{R},\neg\mathcal{F})$ and by precondition, one of the following is true:

1. There exist z, v with $1 \leq z \leq n$ such that $src_{z|\varnothing} \vDash H_v$. Because of $trans \vDash seq$, we have $m_z \vDash src_z$ and, with $m_z : L_z \hookrightarrow G_{z-1}$ and Lemma 2.38 (p. 43), we gain $G_{z-1} \vDash src_{z|\varnothing}$ and, by implication of graph constraints (Definition 2.36 (p. 42)), $G_{z-1} \vDash H_v$.
2. There exists v such that $tar_{n|\varnothing} \vDash H_v$. Because of $trans \vDash seq$, we have $m'_n \vDash tar_n$ and, with $m'_n : R_n \hookrightarrow G_n$ and Lemma 2.38 (p. 43), we gain $G_n \vDash tar_{n|\varnothing}$ and, by implication of graph constraints (Definition 2.36 (p. 42)), $G_n \vDash H_v$.
3. There exists v such that $src_{1|\varnothing} \vDash S_v$. Because of $trans \vDash seq$, we have $m_1 \vDash src_1$ and, with $m_1 : L_1 \hookrightarrow G_0$ and Lemma 2.38 (p. 43), we gain $G_0 \vDash src_{0|\varnothing}$ and, by implication of graph constraints (Definition 2.36 (p. 42)), $G_0 \vDash SC_v$.

Hence, all transformation sequences of length between 0 and k leading to F have violations of $\mathcal{H}$ in any graph or $\mathcal{S}$ in the leftmost graph. Then, $\mathcal{F}$ is satisfied by all graphs in the $k-1$-bounded state spaces of graph grammars induced by GTS and $\mathcal{S}$. □

The differences between $k-1$-bounded backward model checking in the general approach (Theorem T.3g (p. 101)) and the restricted approach (here) are similar to the differences for k-inductive invariant checking mentioned in Section 6.4. Again, a composed forbidden pattern is split into individual forbidden patterns and intermediate source and target patterns are analyzed instead of a leftmost source pattern. Theorem T.3r establishes a sufficient condition, not an equivalence, for the validity of the composed forbidden pattern in bounded state spaces. Any result from the general approach implies the same result with the restricted approach, but false negatives in the restricted approach would have been discarded (if the procedure terminates) by the general approach.

As before, we require $\mathcal{S} \vDash \mathcal{F}$. We have discussed that this can be achieved by choosing the respective composed patterns appropriately. Given an implementation of the implication check described by Theorem 6.8 (p. 120), we can also verify this property.

In comparison to the verification of k-inductive invariants (Theorem T.2r (p. 143)) for the restricted approach, the following differences and similarities stand out:

First, $k-1$-bounded backward model checking requires computation of $\mathrm{SEQ}^r_{k-1}(\mathcal{R},\neg\mathcal{F})$ rather than $\mathrm{Seq}^r_k(\mathcal{R},\neg\mathcal{F})$. By definition of k-bounded state spaces under a constraint, any graph reachable by transformation sequences of length $k-1$ or less without violations of the composed guaranteed pattern belongs to the state space. Thus, we have to analyze shorter sequences, too.

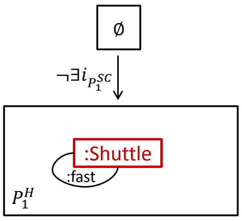


Figure 6.19. – Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$

Second, conditions (1) and (2) are very similar to Theorem T.2r (p. 143). In both cases, the violation of guaranteed constraints allows us to discard the s/t-pattern sequences as a potential counterexample because we analyze both inductive invariants (Definition D.1 (p. 62)) and bounded state spaces (Definition 4.1 (p. 55)) under the assumption of a guaranteed constraint. However, as opposed to inductive invariant checking, we do not check the reduced source pattern for implication of forbidden patterns (condition (1)); encountering a forbidden pattern in an intermediate pattern in the s/t-pattern sequence does not invalidate the sequence as a counterexample.

Finally, condition (3) ensures that s/t-pattern sequences are only considered as counterexamples if their satisfying transformation sequences originate in valid start graphs – i.e. in graphs satisfying the composed start configuration pattern $\mathcal{S}$. If the leftmost (reduced) pattern in an s/t-pattern sequence violates at least one of the start configuration patterns, no satisfying transformation starts from a valid start graph. If the leftmost (reduced) source pattern does not imply any of the start configuration patterns, graphs satisfying that reduced pattern may be valid start graphs starting a transformation sequence leading to a violation of $\mathcal{F}$ – within the bounded state space.

The word 'may' in the previous sequence deserves additional attention: as with Theorem T.2r (p. 143), Theorem T.3r may have false negatives as its results. Here – similar to before – those will be s/t-pattern sequences whose satisfying transformation sequences will not fulfill the criterion of violating $\mathcal{F}$ in the bounded state space (under the composed guaranteed pattern). In particular, an s/t-pattern sequence may have only satisfying transformation sequences that, while indeed leading to $\neg\mathcal{F}$, will have violations of $\mathcal{H}$ at some point or will not have $\mathcal{S}$ satisfied in its leftmost graph. We will discuss the problem of false negatives in Section 6.8.

Example 6.21 (1-bounded backward model checking for an unsafe system)**.** Consider the composed forbidden pattern $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$ (Figure 6.16(d), p. 148), the set of graph rules $\mathcal{R} = \{\mathsf{s2s}, \mathsf{a2b}, \mathsf{f2b}, \mathsf{b2s}, \mathsf{f2f}, \mathsf{s2f}, \mathsf{a2f}\}$ as before (Examples 6.1 (p. 111) and 6.20 (p. 146)), and the composed guaranteed pattern $\mathcal{H} = \bigwedge_{1\leq i\leq 15} \neg H_i$ (Examples 6.1 (p. 111) and 6.20 (p. 146)).

Also, we choose a start configuration constraint $\mathcal{S} = \neg SC_1 = \neg\exists i_{P_1^{SC}}$ (Figure 6.19) that allows all start configurations that do not contain a shuttle driving in speed mode fast. Note that $\mathcal{S} \vDash \mathcal{F}$. The general idea behind this start configuration constraint is that system initialization with shuttles already driving fast is not plausible. This example is equivalent to Example 5.19 (p. 103) in our general approach.

Given this system and composed start configuration pattern, we can find a single rule application from a possible start graph leading to a violation. Consider the s/t-pattern sequence $seq \in \mathrm{Seq}_1^r(\mathcal{R}, F_1)$ (here, $\mathrm{SEQ}_{k-1}^r(\mathcal{R}, \neg\mathcal{F}) = \mathrm{Seq}_1^r(\mathcal{R}, F_1)$), which is depicted in Figure 6.20. We have:

$$seq = src_1 \Rightarrow_{\mathsf{a2f}} tar_1 = \exists(s_1, \neg\exists x_1^1 \wedge \neg\exists x_1^2 \wedge \neg\exists x_1^3) \Rightarrow_{\mathsf{a2f}} tar_1,$$

where $\neg\exists x_1^1 \wedge \neg\exists x_1^2 \wedge \neg\exists x_1^3$ is the result of shifting the rule's negative application condition to the context of the source pattern (cf. step SC_1-3 of the Seq-construction).

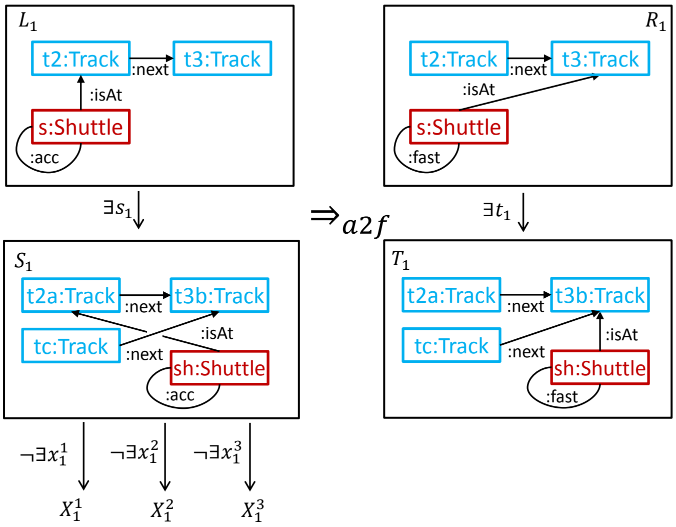


Figure 6.20. – S/t-pattern sequence $seq = src_1 \Rightarrow_{\mathsf{a2f}} tar_1$

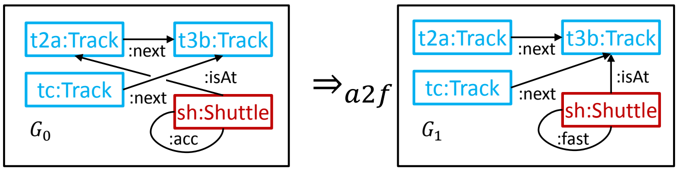
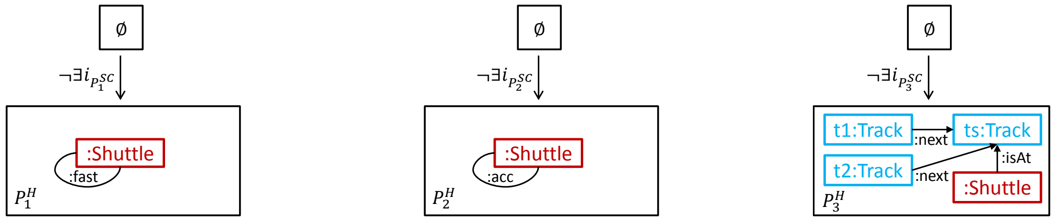


Figure 6.21. – Transformation sequence $trans = G_0 \Rightarrow_{\mathsf{a2f},m_1,m_1'} G_1$

Neither $src_{1|\varnothing}$ nor $tar_{1|\varnothing}$ imply any of the guaranteed patterns H_j, so the potential counterexample cannot be discarded on the basis of the conditions in Theorem T.3r (p. 149). In other words, we have a valid s/t-pattern sequence representing transformation sequences of length leading to a violation of the composed forbidden pattern $\mathcal{F}$. Furthermore, we cannot determine implication of SC_1 by $src_{1|\varnothing}$: the first graph of a satisfying transformation sequence may not violate the start configuration pattern $\mathcal{S}$ and, as a result, may be a valid start graph. Then, we must assume the existence of a satisfying transformation sequence $trans = G_0 \Rightarrow_{\mathsf{a2f}} G_1$ such that $G_0 \vDash \mathcal{S}$, implying that $GG = (GTS, G_0)$ is a graph grammar in $\mathrm{IND}(GTS, \mathcal{S})$ – and that G_1 is contained in the 1-bounded state space under $\mathcal{H}$ ($\mathrm{REACH}(GG, \mathcal{H})$) and violates $\mathcal{F}$. Figure 6.21 depicts such a transformation sequence $trans$ with $trans \vDash seq$, although the construction of explicit transformation sequence is not usually part of the verification process.

Note that in contrast to 2-inductive invariant checking (Example 6.20 (p. 146)), the rule's negative application condition does not help to prevent the violation here. Intuitively, the negative application condition is only able to prevent a violation if the shuttle has sufficient time – in the sense of rule applications – before it arrives at a switch: application of a2f is only forbidden if there is a switch two tracks ahead. However, if the system is initialized with a shuttle positioned directly in front of a switch and in speed mode acc, the negative application is not sufficient.

Since we can find a violation for $\mathcal{F} = \neg F_1$, we will also find violations for the extended safety property $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ with F_2 and F_3 as in Example 6.1 (p. 111) and Example 6.20 (p. 146), Figures 6.18(a) and 6.18(b). These results are expected: we have established in Example 5.18 (p. 101) (general approach) that $\mathcal{F}$ is not valid in the k−1-bounded state space (under $\mathcal{H}$) of all graph grammars induced by GTS and $\mathcal{S} = \neg SC_1$. Necessarily, the restricted approach will come to the same conclusion. △



(a) Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$ **(b)** Constraint $\neg SC_2 = \neg\exists i_{P_2^{SC}}$ **(c)** Constraint $\neg SC_3 = \neg\exists i_{P_3^{SC}}$

Figure 6.22. – Fragments of start configuration constraints

Example 6.22 (1-bounded backward model checking for a safe system)**.** Now, we extend the composed forbidden pattern with $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$; all rules and the composed guaranteed pattern remain unchanged. Figures 6.22(b) and 6.22(c) depict additional start configuration patterns $SC_2 = \exists i_{P_2^{SC}}$ and $SC_3 = \exists i_{P_3^{SC}}$; then, we choose the composed start configuration pattern as $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$.

In Example 6.21 above, we have found a symbolic error trace *seq* (Figure 6.20) from possible start graphs to a violation of $\mathcal{F}$ in the 1-bounded state space (under $\mathcal{H}$) of a graph grammar in IND(GTS,$\mathcal{S}$). In particular, *trans* (Figure 6.21) is a concrete transformation sequence where such a violation occurs. However, with our new composed start configuration pattern, we will find that $src_{1|\varnothing}$ implies SC_2: the reduced source pattern's context contains a shuttle in speed mode acc. Hence, any satisfying transformation sequence (including $trans = G_0 \Rightarrow_{\mathsf{a2f},m_1,m_1'} G_1$) will not originate in a valid start graph. In particular, $G_0 \vDash SC_2$. Also, although not shown here, we can discard any of the remaining s/t-pattern sequence in $\text{SEQ}_1^r(\mathcal{R}, \neg\mathcal{F})$ for similar reasons. Hence, we have $G \vDash \mathcal{F}$ for all graphs G in the 1-bounded state spaces $\text{REACH}_1(GG)$ of all graph grammars in IND(GTS,$\mathcal{S}$).

Intuitively, this makes sense: the interplay between the speed mode protocol (Example 6.1, Figure 6.1(b), p. 112), the rules' composed negative application conditions, and the composed start configuration pattern prevent an early violation of the composed forbidden pattern. This has been discussed in Example 5.19 (p. 103) for the general approach. Here, our restricted approach comes to the same conclusion. △

This concludes our consideration of k-bounded backwards model checking. As with our general approach, k-bounded backwards model checking is – in the context of this thesis – applied to establish the base case for our inductive step, with both elements part of the inductive argument in Lemma L.1 (p. 65). Their combination will be discussed in the following section.

6.6. Operational Invariant Checking

With the verification of k-inductive invariants (Theorem T.2r (p. 143) in Section 6.4) and k−1-bounded state spaces of induced graph grammars (Theorem T.3r (p. 149) in Section 6.5), we can combine both into a single theorem verifying the validity of operational invariants. In other words, we verify the validity of a composed forbidden pattern under a composed guaranteed pattern in all graphs of the state spaces of all graph grammars induced by a graph transformation system and a composed start configuration pattern. As with our general approach (Section 5.5), this combination of Theorem T.2r (p. 143) and Theorem T.3r (p. 149) will result in a new theorem:

Theorem T.4r (operational invariant checking)**.** *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o\in O} \neg S_o$ be a composed forbidden* *pattern, composed guaranteed pattern, and composed start configuration pattern, respectively, with* $\mathcal{S} \vDash \mathcal{F}$.

For all graph grammars $GG = (GTS, G_0)$ *with* $GG \in \mathrm{IND}(GTS, \mathcal{S})$, $\mathcal{F}$ *is an operational invariant of GG under* $\mathcal{H}$ *if there is a* $k \geq 1$ *such that the following conditions hold:*

1. *For all sequences* $seq = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n)$ *with* $seq \in \mathrm{SEQ}^r_k(\mathcal{R}, \neg\mathcal{F})$, *one of the following conditions holds:*

   1.1. $\exists z, v(1 \leq z \leq n \wedge (src_{z|\varnothing} \vDash H_v))$.
   1.2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.
   1.3. $\exists v(src_{1|\varnothing} \vDash SC_v)$.

2. *For all sequences* $\mathrm{prop}(seq) = (src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n)$ *with* $seq \in \mathrm{Seq}^r_k(\mathcal{R}, \neg\mathcal{F})$, *one of the following conditions holds:*

   2.1. $\exists z, v(1 \leq z \leq k \wedge (src_{z|\varnothing} \vDash H_v \vee src_{z|\varnothing} \vDash F_v))$.
   2.2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.

Proof. By precondition and Theorem T.3r (p. 149), we have $G \vDash \mathcal{F}$ for all graphs $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ and all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$. By precondition and Theorem T.2r (p. 143), $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ for GTS. Then, by Lemma 4.12 (p. 63), all graphs $G \in \mathrm{REACH}(GG, \mathcal{H})$ for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$ satisfy $\mathcal{F}$, concluding the proof. □

This theorem is a combination of Theorems T.2r (p. 143) and T.3r (p. 149); as such, the differences between the general and restricted approaches for inductive invariant checking and bounded backward model checking apply here, too. In particular, the problem of false negatives in the base case and the inductive step is still relevant. However, there is an additional reason why the theorem only establishes a sufficient condition: even if there are s/t-pattern sequences representing potential violations of $\mathcal{F}$ as a k-inductive invariant, we cannot be sure that these particular violations occur in the graph grammars' state spaces. It is entirely possible that the start of a transformation sequence violating the k-inductive invariant is not reachable from the allowed start graphs. However, our verification technique would not be able to detect this inherent problem. The approach, by design, only considers reachability to a limited degree.

On the other hand, for a violation of $\mathcal{F}$ in the $k-1$-bounded state space, if established by condition (1) – Theorem T.3r (p. 149) – the situation is clearer: if it is indeed a true negative, then the violation will actually occur.

Example 6.23 (operational invariant checking for an unsafe system)**.** Examples for systems where the composed forbidden pattern $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ cannot be established as an operational invariant via Theorem T.4r are Example 6.19 (p. 144) and Example 6.21 (p. 151).

In the former example, $\mathcal{F}$ is not a 2-inductive invariant or, although not shown there, an invariant for any value of k. However, without further analysis, we cannot be sure $\mathcal{F}$ is not an operational invariant; i.e. that there exists a graph in a graph grammar's state space violating $\mathcal{F}$. As mentioned before, Theorem T.4g does not provide a necessary condition to verify operational invariants.

In the latter example, there is a transformation sequence (Figure 6.21) leading to a violation of $\mathcal{F}$ after a rule application to a possible start graph. Hence, $\mathcal{F}$ is not an operational invariant for the system described in that example.

This example is identical to Example 5.20 (p. 105) in our general approach. △

Example 6.24 (operational invariant checking for a safe system)**.** In Examples 6.20 (p. 146) and 6.22 (p. 152), we have used a system with the rule set $\mathcal{R} = \{$f2f, a2f, s2a, s2s, f2b, a2b, b2s$\}$, the composed forbidden pattern $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$, a composed guaranteed pattern $\mathcal{H} =$

$\bigwedge_{1 \leq i \leq 15} \neg H_i$, and a composed start configuration pattern $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3$. We have shown via Theorems T.2r (p. 143) and T.3r (p. 149) that $\mathcal{F}$ is a 2-inductive invariant for $GTS = (TG, \mathcal{R})$ under $\mathcal{H}$ and that $\mathcal{F}$ is valid in all 1-bounded state spaces of graph grammars in $\mathrm{IND}(GTS, \mathcal{S})$. In combination and by Theorem T.4r (p. 153), $\mathcal{F}$ is satisfied in all graphs of all state spaces of the induced graph grammars under the constraint $\mathcal{H}$. This example is identical to Example 5.21 (p. 105) in the general approach. △

Similar to our general approach, Theorem T.4r (p. 153) provides a constructive approach to verifying Lemma L.1 (p. 65). However, the theorem only considers state spaces under a composed guaranteed pattern $\mathcal{H}$. In other words, we assume the composed guaranteed pattern's validity in the systems at hand. To consider explicit verification of $\mathcal{H}$, we recall the respective lemma from Chapter 4:

Lemma L.2 (validity of constraints in induced graph grammars under a constraint [4]). *Let $GG = (GTS, G_0)$ be a graph grammar with a graph transformation system $GTS = (TG, \mathcal{R})$ and let $\mathcal{F}$, $\mathcal{H}$, and $\mathcal{S}$ be graph constraints with $\mathcal{S} \vDash \mathcal{F}$ and $\mathcal{S} \vDash \mathcal{H}$. $\mathcal{F}$ is an operational invariant of* $\mathrm{REACH}(GG)$ *for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$, if the following conditions hold:*

0. $\mathcal{H}$ is a 1-inductive invariant for GTS.

1/2. $\mathcal{F}$ is an operational invariant of GG under $\mathcal{H}$ for all graph grammars $GG \in \mathrm{IND}(GTS, \mathcal{S})$.

By verifying the validity of $\mathcal{H}$ as a 1-inductive invariant and in all possible start graphs (by $\mathcal{S} \vDash \mathcal{H}$), we can show the equivalence of state spaces under $\mathcal{H}$ and state spaces without that restriction. The additional requirement of $\mathcal{S} \vDash \mathcal{H}$ can be achieved as discussed before: we can choose a desired composed start configuration pattern $\mathcal{S}'$ and define $\mathcal{S} = \mathcal{S}' \wedge \mathcal{H}$. We can also check implication via Theorem 6.8 (p. 120).

The following theorem uses the results of this chapter to extend Theorem T.4r (p. 153) to include verification of $\mathcal{H}$:

Theorem T.5r. *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o \in O} \neg SC_o$ be a composed forbidden pattern, composed guaranteed pattern, and composed start configuration pattern, respectively, with $\mathcal{S} \vDash \mathcal{F}$ and $\mathcal{S} \vDash \mathcal{H}$.*

For all graph grammars $GG = (GTS, G_0) \in \mathrm{IND}(GTS, \mathcal{S})$, $\mathcal{F}$ is an operational invariant of GG if the following conditions hold:

0. For all sequences $seq = (src_1 \Rightarrow_{b_1} tar_1)$ with $seq \in \mathrm{Seq}_1^r(\mathcal{R}, \neg\mathcal{H})$ the following condition holds:

0.1. $\exists z, v(1 \leq z \leq n \wedge (src_{1|\varnothing} \vDash H_v))$.

1. For all sequences $seq = (src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_n} tar_n)$ with $seq \in \mathrm{SEQ}_k^r(\mathcal{R}, \neg\mathcal{F})$, one of the following conditions holds:

1.1. $\exists z, v(1 \leq z \leq n \wedge (src_{z|\varnothing} \vDash H_v))$.

1.2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.

1.3. $\exists v(src_{1|\varnothing} \vDash SC_v)$.

2. For all sequences $\mathrm{prop}(seq) = (src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_n} tar_n)$ with $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$, one of the following conditions holds:

2.1. $\exists z, v(1 \leq z \leq k \wedge (src_{z|\varnothing} \vDash H_v \vee src_{z|\varnothing} \vDash F_v))$.

2.2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.

Proof. With (0) and by Theorem T.2r (p. 143) – and appropriate subsitutions – $\mathcal{H}$ is a 1-inductive invariant for GTS. By Theorem T.2g (p. 96) and (2), $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$. By Theorem T.3g (p. 101) and (1), $\mathcal{F}$ is valid in all k−1-bounded state spaces of graph grammars in $\mathrm{IND}(GTS, \mathcal{S})$ under $\mathcal{H}$. By Lemma L.2, we get $G \vDash \mathcal{F}$ for all graphs G with $G \in \mathrm{REACH}(GG)$ and $GG \in \mathrm{IND}(GTS, \mathcal{S})$. □

Conditions (1) are (2) are merely repeated from Theorem T.4r (p. 153) and establish $\mathcal{F}$ as an oeprational invariant under $\mathcal{H}$. Condition (0) verifies $\mathcal{H}$ as a 1-inductive invariant for GTS and, given $\mathcal{S} \models \mathcal{H}$, as an operational invariant. Then, REACH(GG) and REACH$(GG, \mathcal{H})$ are identical per induced graph grammar $GG \in \text{IND}(GTS, \mathcal{S})$. As a result, $\mathcal{F}$ is an operational invariant even without restricting the state space by $\mathcal{H}$.

As in our general approach, we can decouple verification of $\mathcal{F}$ under $\mathcal{H}$ and verification of $\mathcal{H}$. Depending on the type of composed guaranteed pattern, this may or may not be applicable in the application scenario at hand. External assumptions, for example, are not usually verifiable as inductive or operational invariants. Again, a typical example is a single fault assumption.

Example 6.25 (operational invariant checking and composed guaranteed pattern)**.** Analogously to Example 5.22 (p. 106) in the general approach, we consider Example 6.24 (p. 154) with an extended composed start configuration pattern $\mathcal{S} = \neg SC_1 \wedge \neg SC_2 \wedge \neg SC_3 \wedge \mathcal{H}$. Then, $\mathcal{S} \models \mathcal{H}$ as required by Theorem T.5r. The results with respect to conditions (1) and (2) of Theorem T.4r (p. 153) and condition (1/2) of Lemma L.2 still hold: $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ and $\mathcal{F}$ is valid in every graph in REACH$_{k-1}(GG, \mathcal{H})$ for graph grammars $GG \in \text{IND}(GTS, \mathcal{S})$ – for both the former and extended composed start configuration pattern. Also, although not shown here, $\mathcal{H}$ is a 1-inductive invariant of GTS. Then, by Theorem T.5r (p. 155), $\mathcal{F}$ is an operational invariant of REACH(GG) for all induced graph grammars $GG \in \text{IND}(GTS, \mathcal{S})$. △

6.7. Implementation

Implementation of the constructions and theorems described in the previous sections is roughly separated with respect to whether they relate to k-inductive invariant checking (as in Section 6.4), k−1-bounded backward model checking (as in Section 6.5), or their combination (as in Section 6.6).

6.7.1. k-inductive Invariant Checking

First, we will discuss the implementation of k-inductive invariant checking.

Algorithm 6.2: k-invcheck

```
desc.  : basic verification scheme for $k$-inductive invariant checking (Theorem T.2r)
input  : an integer $k$ with $k \geq 1$, a set $\mathcal{R}$ of graph rules, a set $\mathcal{F}$ of forbidden
         patterns, a set $\mathcal{H}$ of guaranteed patterns
output: a set of $k$-sequences of $s/t$-patterns as counterexamples
results ← ∅
sequences ← ∅
foreach $F \in \mathcal{F}$ do                                        /* $\text{Seq}_1^r(\mathcal{R}, F)$ */
    sequences ← sequences ∪ createSequences($\mathcal{R}, F$)
for $i \leftarrow 2$ to $k$ do                                         /* $\text{Seq}_k^r(\mathcal{R}, F)$ */
    sequences ← extendSequences($\mathcal{R}$, sequences)
foreach seq ∈ sequences do                                      /* analysis */
    if not discardSequence(seq, $\mathcal{F}, \mathcal{H}$) then
        results ← results ∪ {seq}
return results
```

```
Algorithm 6.3: createSequences($\mathcal{R}$, F)
desc.   : implements $\mathrm{Seq}^r_1(\mathcal{R}, F)$
input   : a set $\mathcal{R}$ of graph rules, a forbidden pattern $F = \exists(i_P, ac_P)$
output: $\mathrm{Seq}^r_1(\mathcal{R}, F)$
$results \leftarrow \varnothing$
foreach $b \in \mathcal{R}$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ do
    foreach $tar$ in $\bigvee_{j \in J} tar_j = \mathrm{Shift}(i_R, F)$ do                /* SC$_1$-1 */
        $src' \leftarrow \mathrm{L}(b, tar)$                                          /* SC$_1$-2 */
        if $src'$ has the form $\exists(s, ac')$ then                              /* SC$_1$-3 */
            $src \leftarrow \exists(s, ac' \wedge \mathrm{Shift}(s, ac_L \wedge \mathrm{Appl}(b)))$           /* SC$_1$-3 */
            $results \leftarrow results \cup \{src \Rightarrow_b tar\}$                   /* SC$_1$-4 */
return $results$                                                          /* SC$_1$-5 */
```

```
Algorithm 6.4: extendSequences($\mathcal{R}$, $sequences$)
desc.   : implements $\mathrm{Seq}^r_{k+1}(\mathcal{R}, \neg\mathcal{F})$, based on the result of $\mathrm{Seq}^r_k(\mathcal{R}, \neg\mathcal{F}))$
input   : $\mathrm{Seq}^r_k(\mathcal{R}, F)$, i.e. a set of sequences $sequences$ of the form
          $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ with $src_1 = \exists(s_1, ac_{S_1})$, a set $\mathcal{R}$ of graph rules
output: $\mathrm{Seq}^r_{k+1}(\mathcal{R}, F)$
$results \leftarrow \varnothing$
foreach $seq \in sequences$ with $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ do
    foreach $b \in \mathcal{R}$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ do
        foreach $tar$ in $\bigvee_{j \in J} tar_j = \mathrm{Shift}(i_R, src_{1|\varnothing})$
         with $tar = \exists(t, ac_T)$ do                                          /* SC$_k$-1 */
            $src_1^+ \leftarrow \exists(s' \circ s_1, ac_T)$                                 /* SC$_k$-1$^+$ */
            $src' \leftarrow \mathrm{L}(b, tar)$                                      /* SC$_k$-2 */
            if $src'$ has the form $\exists(s, ac')$ then                          /* SC$_k$-3 */
                $src \leftarrow \exists(s, ac' \wedge \mathrm{Shift}(s, ac_L \wedge \mathrm{Appl}(b)))$       /* SC$_k$-3 */
                $results \leftarrow results \cup \{src \Rightarrow_b (tar, src_1^+) \Rightarrow_{b_1} \dots \Rightarrow tar_k\}$   /* SC$_k$-4 */
return $results$                                                          /* SC$_k$-5 */
```

```
Algorithm 6.5: discardSequence($seq$, $\mathcal{F}$, $\mathcal{H}$)
desc.   : implements conditions (1) and (2) of Theorem T.2r (p. 143)
input   : a $k$-sequence of $s/t$-patterns $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$, a set $\mathcal{F}$ of forbidden
          patterns, a set $\mathcal{H}$ of guaranteed patterns
output: whether $seq$ contains (reduced) patterns that imply patterns in $\mathcal{F}$ or $\mathcal{H}$
if discardPattern($tar_{k|\varnothing}$, $\mathcal{H}$) then                            /* condition (2) */
    return true
for $i \leftarrow 1$ to $k$ do                                            /* condition (1) */
    if discardPattern($src_{i|\varnothing}$, $\mathcal{F} \cup \mathcal{H}$) then
        return true
return false
```

Algorithm 6.6: discardPattern(C, $\mathcal{C}$)

```
desc.  : applies implication check of Theorem 6.8 (p. 120)
input  : a graph pattern $C$, a set $\mathcal{C}$ of forbidden/guaranteed patterns
output: whether $C$ implies any pattern in $\mathcal{F}$ or $\mathcal{H}$

foreach $C' \in \mathcal{C}$ do
    if implies($C, C'$) then                    /* Theorem 6.8, Algorithm 6.1 */
        return true
return false
```

A basic scheme implementing Theorem T.2r (p. 143) is shown in Algorithm 6.2. The functions createSequences and extendSequences refer to the implementation of the Seq-construction and are described in Algorithms 6.3 and 6.4.

The function discardSequence (Algorithm 6.5) is an implementation of conditions (1) and (2) in Theorem T.2r (p. 143); its task is to discard invalid counterexamples. Its prerequisite discardPattern is shown in Algorithm 6.6. It is based on implication of graph patterns described by Theorem 6.8 (p. 120) and Algorithm 6.1 (p. 121).

However, this naive implementation is rather inefficient: it will first create all s/t-pattern sequences of length k, then discard sequences with violations. Instead, it is possible to interweave the creation and extension of sequences with the analysis for violations of forbidden or guaranteed patterns. To achieve this, we can refine the implementation of createSequences and extendSequences as described in Algorithms 6.8 and 6.9. There, code lines performing intermediate analysis of source or target patterns are annotated accordingly. Then, separate analysis is not required; the overall algorithm is shown in Algorithm 6.7.

Algorithm 6.7: k-invcheck

```
desc.  : verification scheme for $k$-inductive invariant checking (Theorem T.2r)
input  : an integer $k$ with $k \geq 1$, a set $\mathcal{R}$ of graph rules, a set $\mathcal{F}$ of forbidden
         patterns, a set $\mathcal{H}$ of guaranteed patterns
output: a set of $k$-sequences of $s/t$-patterns as counterexamples

$results \leftarrow \emptyset$
foreach $F \in \mathcal{F}$ do                                    /* $\mathrm{Seq}_1^r(\mathcal{R}, F)$ */
    $results \leftarrow results \cup$ createSequences($\mathcal{R}, F, \mathcal{H}, \mathcal{F} \cup \mathcal{H}$)
for $i \leftarrow 2$ to $k$ do                                     /* $\mathrm{Seq}_k^r(\mathcal{R}, F)$ */
    $results \leftarrow$ extendSequences($\mathcal{R}, results, \mathcal{F} \cup \mathcal{H}$)
return $results$
```

Intermediate anaylsis in createSequences (Algorithm 6.8, which implements $\mathrm{Seq}_1^r(\mathcal{R}, F)$) happens in three places:

First, after the existential condition $\exists i_P$ of a forbidden pattern $F = \exists(i_P, ac_P)$ has been shifted to the right rule side, the resulting context of new target patterns $\exists(t : R \hookrightarrow T)$ is checked for implication of guaranteed patterns. The idea here is that some target patterns can be discarded without taking their nested composed negative application condition into account: if T contains a graph P' that occurs in a guaranteed pattern $\exists i_{P'}$ with a trivial composed negative application condition (true), the target pattern can be discarded. In the context of Theorem T.2r (p. 143), this implements a part of condition (2).

Second, after transferring the forbidden pattern's composed negative application condition

Algorithm 6.8: createSequences($\mathcal{R}$, F, $\mathcal{C}_1$, $\mathcal{C}_2$)

desc. : implements $\mathrm{Seq}_1^r(\mathcal{R}, F)$ with optimizations

input : a set $\mathcal{R}$ of graph rules, a forbidden pattern $F = \exists(i_P, ac_P)$, sets $\mathcal{C}_1$ and $\mathcal{C}_2$ of patterns

output: $\mathrm{Seq}_1^r(\mathcal{R}, F)$ minus sequences discarded for violations of $\mathcal{C}_1$ or $\mathcal{C}_2$

$$\begin{tikzcd}
L \arrow[d, hook, "s"'] \arrow[dr, phantom, "\triangledown ac_L" near start] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] & R \arrow[d, hook, "t"'] \arrow[dr, phantom, "ac_P \vartriangle" near end] & \varnothing \arrow[l, hook', "i_R"'] \arrow[d, hook, "i_P"] \arrow[dl, phantom, "="] \\
S \arrow[ur, phantom, "\vartriangle ac'_S" near start] & D \arrow[l, hook'] \arrow[r, hook] \arrow[ur, phantom, "ac_T \vartriangle" near end] & T & P \arrow[l, hook', "p"']
\end{tikzcd}$$

```
results ← $\varnothing$
foreach $b \in \mathcal{R}$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ do
    foreach tar in $\bigvee_{j \in J} tar_j = \mathrm{Shift}(i_R, \exists i_P)$ with $tar = \exists t$ do      /* SC$_1$-1 */
        if not discardPattern($tar_{|\varnothing}$, $\mathcal{C}_1$) then                                /* analysis */
            $tar \leftarrow \exists(t, \mathrm{Shift}(p, ac_P))$                                        /* SC$_1$-1 */
            if not discardPattern($tar_{|\varnothing}$, $\mathcal{C}_1$) then                            /* analysis */
                $src' \leftarrow \mathrm{L}(b, tar)$                                                    /* SC$_1$-2 */
                if $src'$ has the form $\exists(s, ac')$ then                                          /* SC$_1$-3 */
                    $src \leftarrow \exists(s, ac' \wedge \mathrm{Shift}(s, ac_L \wedge \mathrm{Appl}(b)))$   /* SC$_1$-3 */
                    if not discardPattern($src_{|\varnothing}$, $\mathcal{C}_2$) then                    /* analysis */
                        $results \leftarrow results \cup \{src \Rightarrow_b tar\}$                     /* SC$_1$-4 */
return results                                                                                         /* SC$_1$-5 */
```

ac_P to the context of the target pattern – $tar = \exists(t, \mathrm{Shift}(p, ac_P))$ – $tar_{|\varnothing}$ is checked again. Now, it may imply guaranteed patterns with non-trivial composed negative application conditions. Deferring the transfer of a forbidden pattern's composed negative application condition to the target pattern in question is not considered in the Seq-construction described in Theorem T.1r (p. 130) – it is an optimization specific to the implementation. Note that $tar_{|\varnothing} \vDash H$ for guaranteed patterns H with a trivial composed negative application condition (true) does not have to be checked again; the result will be the same as in the first intermediate check above. In the context of Theorem T.2r (p. 143), this implements the remaining part of condition (2).

Third, the resulting (reduced) source pattern is analyzed for implication of guaranteed or forbidden patterns. In the context of Theorem T.2r (p. 143), this implements the part of condition (1) where src_k is concerned.

Algorithm 6.9: extendSequences($\mathcal{R}$, *sequences*, $\mathcal{C}$)

desc. : implements $\mathrm{Seq}^r_{k+1}(\mathcal{R}, \neg\mathcal{F})$ (based on $\mathrm{Seq}^r_k(\mathcal{R}, \neg\mathcal{F})$) with optimizations

input : $\mathrm{Seq}^r_k(\mathcal{R}, F)$, i.e. a set of sequences *sequences* of the form $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$ with $src_1 = \exists(s_1, ac_{S_1})$, a set $\mathcal{R}$ of graph rules, a set $\mathcal{C}$ of patterns

output: $\mathrm{Seq}^r_{k+1}(\mathcal{R}, F)$ minus sequences discarded for violations of $\mathcal{C}$

$$\begin{tikzcd}
L \arrow[d, hook, "s"] \arrow[dr, phantom, "\triangledown ac_L"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] & R \arrow[d, hook, "t"] & L_1 \arrow[d, hook, "s_1"] \\
S \arrow[ur, phantom, "\triangle ac'_S"] & D \arrow[l, hook'] \arrow[r, hook] \arrow[r, phantom, "ac_T \triangle"] & T & S_1 \arrow[l, hook', "s'"'] \arrow[l, phantom, "ac_{S_1} \triangle"]
\end{tikzcd}$$

```
results ← ∅
foreach seq ∈ sequences with seq = src_1 ⇒_{b_1} ... ⇒_{b_k} tar_k do
    foreach b ∈ R with b = ⟨(L ↩ K ↪ R), ac_L, true⟩ do
        foreach tar in ⋁_{j∈J} tar_j = Shift(i_R, ∃i_{S_1}) with tar = ∃t do      /* SCk-1 */
            if not discardPattern(tar_{|∅}, C) then                              /* analysis */
                ac_T ← Shift(s', ac_{S_1})                                       /* SCk-1 */
                tar ← ∃(t, ac_T)                                                 /* SCk-1 */
                if not discardPattern(tar_{|∅}, C) then                          /* analysis */
                    src_1^+ ← ∃(s' ∘ s_1, ac_T)                                  /* SCk-1+ */
                    src' ← L(b, tar)                                             /* SCk-2 */
                    if src' has the form ∃(s, ac') then                          /* SCk-3 */
                        src ← ∃(s, ac' ∧ Shift(s, ac_L ∧ Appl(b)))              /* SCk-3 */
                        if not discardPattern(src_{|∅}, C) then                  /* analysis */
                            results ← results ∪
                            {src ⇒_b (tar, src_1^+) ⇒_{b_1} ... ⇒ tar_k}         /* SCk-4 */
return results                                                                   /* SCk-5 */
```

In extendSequences (Algorithm 6.9), which implements $\mathrm{Seq}^r_{k+1}(\mathcal{R}, \neg\mathcal{F})$, intermediate analysis happens in a similar fashion. The main difference is that the created target patterns have to be analyzed for implication of both guaranteed or forbidden patterns. The three steps implement different parts of condition (1) of Theorem T.2r. Note that for each target/source pattern (tar_i, src^+_{i+1}), we have $tar_{i|\varnothing} = src^+_{i+1|\varnothing}$ – hence the ostensible difference in pattern analysis between Theorem T.2r (p. 143) and its implementation.

In practice, the separation between the positive part of a newly created target pattern and its composed negative application condition may have a significant impact on performance. Shifting a forbidden pattern's or a source pattern's composed negative application condition to the new target pattern is costly. Target patterns that can be discarded without this computation step reduce the effort of both completing the target pattern in question and of further computations that would involve that target pattern.

These optimized algorithms have similarities to the general version of the Seq-construction (Theorem T.1g (p. 85)). There, both the safety and guaranteed constraint were included in the construction of s/t-pattern sequences. However, both constraints were shifted to all source and target patterns of the sequence, resulting in an explosion in complexity. Here, source and target patterns are analyzed for their implication of individual forbidden or guaranteed patterns, which requires far less computational effort. Furthermore, the s/t-pattern itself is not enhanced – it is either discarded or kept.

6.7.2. k–1-bounded Backward Model Checking

For the implementation of k–1-bounded backward model checking (Theorem T.3r (p. 149)), we can use an adjusted version of Algorithm 6.7, which is shown in Algorithm 6.10. In contrast to Theorem T.2r (p. 143), we have to consider all s/t-pattern sequences from lengths 1 to $k-1$, not just those of length k. We can still discard all s/t-pattern sequences whose (reduced) source or target patterns violate any guaranteed patterns. However, we cannot discard sequences that have violations of the composed start configuration pattern $\mathcal{S}$ because those sequences might still be extended to form a violating sequence.

Therefore, after iteration of each value of i (from 1 to k – 1), we analyze the leftmost source pattern of the resulting s/t-pattern sequences. If its reduction does not imply any start configuration pattern, i.e. violate the composed start configuration pattern, it represents a potential start configuration and the corresponding s/t-pattern sequence is added to the result of symbolic counterexamples.

Algorithm 6.10: k–1-modelcheck

desc. : verification scheme for k–1-bounded backward model checking (Theorem T.3r)

input : an integer k with $k \geq 2$, a set $\mathcal{R}$ of graph rules, a set $\mathcal{F}$ of forbidden patterns, a set $\mathcal{H}$ of guaranteed patterns, a set $\mathcal{S}$ of start configuration patterns

output: a set of s/t-pattern sequences as counterexamples

```
sequences ← $\varnothing$
foreach $F \in \mathcal{F}$ do                                                /* $\mathrm{Seq}^r_1(\mathcal{R}, F)$ */
    sequences ← sequences ∪ createSequences($\mathcal{R}, F, \mathcal{H}, \mathcal{H}$)
    foreach seq ∈ sequences with seq = $src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n$ do      /* analysis */
        if not discardPattern($src_{1|\varnothing}, \mathcal{S}$) then
            results ← results ∪ {seq}
for $i \leftarrow 2$ to $k-1$ do                                              /* $\mathrm{Seq}^r_{k-1}(\mathcal{R}, F)$ */
    sequences ← extendSequences($\mathcal{R}$, sequences, $\mathcal{H}$)
    foreach seq ∈ sequences with seq = $src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_i} tar_i$ do      /* analysis */
        if not discardPattern($src_{1|\varnothing}, \mathcal{S}$) then
            results ← results ∪ {seq}
return results
```

6.7.3. Operational Invariant Checking

Because of the different requirements of k-inductive invariant checking and $k-1$-bounded backward model checking, a combination of both in the sense of Theorem T.4r (p. 153) involves separate execution of Algorithms 6.7 (p. 158) and 6.10 (p. 161), although a combination is, in theory, possible. Furthermore, implementation of Theorem T.5r (p. 155) involves execution of Algorithms 6.7 (p. 158) and 6.10 to establish $\mathcal{F}$ as an operational invariant under $\mathcal{H}$. Then, $\mathcal{H}$ has to be verified as a 1-inductive invariant, which can be achieved by execution of Algorithm 6.7 (p. 158) where the parameter $\mathcal{F}$ is set to $\mathcal{H}$, the parameter $\mathcal{H}$ is set to the empty set of graph patterns $\varnothing$, and k is set to 1. In other words, instead of verifying $\mathcal{F}$ under $\mathcal{H}$ as a k-inductive invariant, we need to verify $\mathcal{H}$ under the trivial guaranteed constraint true as a 1-inductive invariant (cf. condition (0) of Theorem T.5r (p. 155)).

6.8. Discussion and Conclusion

The implementation of Theorems T.2r-T.4r – and hence, of the underlying verification approach outlined by Lemma L.1 (p. 65) – uses a fixed value for k. A stronger requirement would be to leave k unspecified and simply ask whether there exists a k such that the respective conditions are fulfilled. However, there are graph constraints that are not k-inductive invariants for any value of k. Then, an algorithm attempting to find such a k might not terminate, unless it could prove its absence. Given an approach solving the questions above for a specific value of k, however, we can increase that value, if required, and simulate searching for a k while still having the option of terminating the search at a certain threshold.

Table 6.4. – Comparison of general and restricted formal model

Element/ model	Meta-models	System states	Systems	System sets	System state spaces
General formal model	Type graphs	Typed graphs	Typed graph grammars	Induced (typed) graph grammars	Graph grammars' state spaces under guaranteed constraint
Restricted formal model	Type graphs	Typed graphs	Typed graph grammars	Induced (typed) graph grammars	Graph grammars' state spaces under guaranteed constraint

Element/ model	System behavior	Safety properties	Guaranteed properties	Initial states
General formal model	Graph rules with left nested application conditions	Nested graph constraint(s)	Nested graph constraint(s)	Nested graph constraint(s)
Restricted formal model	Graph rules with left composed negative application conditions	Composed graph pattern	Composed graph pattern	Composed graph pattern

The restricted approach is based on the restricted formal model reintroduced earlier. Table 6.4 shows (again) a comparison of the general and restricted formal models. In essence, constraints are to be specified as composed graph patterns instead of nested graph constraints. Graph rules may only have (left) composed negative application conditions, not application

Table 6.5. – Properties of the general and restricted approaches

	Property	**Appl.-soundness**	**Appl.-termination**	**Appl.-deg.-completeness**	**Appl.-performance**
General approach	k-inductive invariant checking	✓	(✓)	?/×	?
	k−1-bounded backward model checking	✓	(✓)	?/×	?
	Operational invariant checking	✓	(✓)	?/×	?
Restricted approach	k-inductive invariant checking	✓	✓	-/×	+
	k−1-bounded backward model checking	✓	✓	-/×	+
	Operational invariant checking	✓	✓	-/×	+

conditions of arbitrary nesting and boolean combinations. These restrictions are part of striking a balance between expressiveness, computational effort, and the impact that the requirement of termination and its implementation have on completeness.

Table 6.5 shows the properties of the restricted approach with respect to the four requirements **Appl.-soundness**, **Appl.-termination**, **Appl.-deg.completeness**, and **Appl.-performance**. It also highlights differences to the general approach. Here, we compare the restricted approach and its implementation to the hypothetical implementation of the general approach mentioned in Section 5.6 and Table 5.3 (p. 108).

By its formalization and implementation, the restricted approach is sound: Theorems T.2r (p. 143), T.3r (p. 149), and T.4r (p. 153) describe sufficient conditions to establish the respective proof goals or find symbolic representations of all counterexamples. The approach always terminates (given a fixed k). Both termination and soundness apply to k-inductive invariant checking, k−1-bounded backward model checking, and operational invariant checking.

There is a subtle difference to the general approach with respect to termination: in the general approach, we included an undecidable problem in the analysis of s/t-pattern sequences – specifically, finding a satisfying transformation sequence. An implementation would have to take additional measures to ensure termination, such as enforcing a threshold on runtime and aborting without a definitive result. For the restricted approach, the analysis of s/t-pattern sequences consists of decidable questions, provided implication of graph patterns is checked via the sufficient condition described by Theorem 6.8 (p. 120). In other words, we get a definitive answer – but the result may be incomplete in the sense of overlooking false negatives, which is discussed in more detail below.

Even without evaluation of case studies, we expect the restricted approach to fare worse then the general approach (if implemented) in terms of completeness (as indicated in Table 6.5). However, we can expect far better performance for two reasons: First, the restrictions on graph constraints and application conditions of the restricted formal model reduce the components' complexity and the computational effort of the Shift-construction in the construction of s/t-pattern sequences. Second, intermediate violations of guaranteed or forbidden patterns in a sequence are not checked by shifting those constraints to the sequences. Instead, they are directly compared to source and target patterns in an s/t-pattern sequence on an individual basis. This is both the reason for an expected improvement in performance – and the main cause of incompleteness.

Table 6.6. – True negatives and true positives in the context of Theorems T.2r or T.3r

Term	Description	Occurrence
True negatives	Counterexamples (s/t-pattern sequences) that do have satisfying transformation sequences as counterexamples in the sense of Lemma 6.10 or Lemma 6.11	Yes, if system is unsafe
True positives	Composed forbidden patterns correctly declared a k-inductive invariant or valid in the bounded state space in the sense of Lemma 6.10 or Lemma 6.11	Yes, if system is safe

Table 6.7. – False negatives and false positives in the context of Theorems T.2r or T.3r

Term	Description	Occurrence
False negatives	Counterexamples (s/t-pattern sequences) that do not have satisfying transformation sequences as counterexamples in the sense of Lemma 6.10 or Lemma 6.11	Possibly (approach is incomplete)
False positives	Composed forbidden patterns incorrectly declared a k-inductive invariant or valid in the bounded state space in the sense of Lemma 6.10 or Lemma 6.11	No (approach is sound)

While specific results on the degree of completeness require detailed evaluation, any lack in completeness is detrimental to **Appl.-deg.completeness** of the intended contribution. Arguably, this is the main drawback of the restricted approach. Incompleteness manifests itself in the occurrence of false negatives (cf. Definition 1.7 (p. 8)). As explained before, false negatives can occur during the verification of k-inductive invariants, during $k-1$-bounded backward model checking, and, as a result, during their combination in operational invariant checking. False negatives are s/t-pattern sequences whose satisfying transformation sequences are not counterexamples in the sense of Lemma 6.10 (p. 123) – or Lemma 6.11 (p. 124), respectively.

Besides these, we may also have true negatives – s/t-pattern sequences that do have satisfying transformation sequences describing a violation. A true positive describes a system correctly classified as safe. False positives, however, must not occur: erroneously discarding counterexamples and possibly classifying an unsafe system as safe may have disastrous consequences. Fortunately, we have established that the approach is sound: all counterexamples will indeed be found. Conversely, if the approach cannot safely establish an s/t-pattern sequence's invalidity as a counterexample, it has to be part of the verification result as a counterexample – but may later turn out to be a false negative (after manual inspection, for example). In order to clarify the meaning of the respective terms, Tables 6.6 and 6.7 summarize the interpretations of true negatives, true positives, false negatives, and false positives in the context of our verification approach. Note that there is a difference in cardinality: when succesfully verifying a composed forbidden pattern, the true positive would just be the composed forbidden pattern itself; if the composed forbidden pattern is not a k-inductive invariant, there may be multiple s/t-pattern sequences as counterexamples, i.e. more than one true negative. There may also be multiple false negatives.

False negatives may appear in a number of different forms as results of the following phenomena related to implication of graph patterns (with details to follow below):

Undecided implication of patterns – since the general problem of graph constraint implication is undecidable, Theorem 6.8 (p. 120) describes a sound, but incomplete approach to checking pattern implication. It uses a sufficient, but not necessary condition; in other words, failure to establish implication may not imply its absence.

Unconsidered interactions of patterns – Theorems T.2r (p. 143) and T.3r (p. 149) and Theorem 6.8 (p. 120) only compare two patterns; however, interactions between patterns can result in implication relations not found by comparing individual patterns only.

Insufficient information in patterns – although graphs in a transformation sequence satisfying an s/t-pattern sequence will have violations of guaranteed or forbidden patterns, implication of the respective patterns cannot be confirmed by Theorem 6.8 if the respective source or target pattern is lacking information.

We will consider each class in the following paragraphs.

Undecided implication of patterns. Given patterns C and C', Theorem 6.8 (p. 120) does not allow to conclude $C \nvDash C'$ from its failure to establish $C \vDash C'$. Hence, we cannot be sure whether $C \vDash C'$ or $C \nvDash C'$, but, in order to ensoure soundness, have to assume $C \nvDash' C$. If $C \vDash C'$ does hold, this decision results in false negatives. This class is the rarest of the three; the other two will amount to the majority of cases of false negatives. Cases where neither implication nor its absence can be established by the implementation of Theorem 6.8 (p. 120) will always involve patterns with non-trivial composed negative application conditions (see explanation of the theorem). Even then, failure to establish implication will often imply absence of implication; then, the counterexample is not a spurious one.

Unconsidered interactions of patterns. Both Theorems T.2r (p. 143) and T.3r (p. 149) and Theorem 6.8 (p. 120) only consider implication between two patterns. However, patterns can interact with each other, which affects the result of the analysis.

Consider the following abstract case (with a specific example discussed later in Section 7.4, Example 7.32 (p. 214)): we wonder whether an s/t-pattern sequence $seq = src \Rightarrow_b tar$ with $seq \in \mathrm{Seq}_1^r(\mathcal{R}, \mathcal{F})$ should be discarded as a counterexample for a 1-indcutive invariant $\mathcal{F} = \neg F$ under $\mathcal{H} = \neg H$. By Theorem T.2r, we need to find out whether the reduced source pattern $src_{|\varnothing}$ implies F or H. If we cannot establish $src_{|\varnothing} \vDash F$ (via Theorem 6.8 (p. 120)), we have to assume (to ensure soundness) $src_{|\varnothing} \nvDash F$: there exists (at least) one graph G with $G \vDash src_{|\varnothing}$ and $G \vDash \neg F$. If we cannot show $src_{|\varnothing} \vDash H$ either, we also need to assume the existence of a graph G' with $G' \vDash src_{|\varnothing}$ and $G' \vDash \neg H$. If we stop the analysis there – which is what Theorems T.2r (p. 143) and 6.8 (p. 120) do – we cannot discard $seq = src \Rightarrow_b tar$ (assuming similar results for the target pattern tar, which we skipped here). However, by focusing on individual patterns, we have overlooked that, in general,

$$\exists G(G \vDash src_{|\varnothing} \wedge G \vDash \neg F) \tag{1}$$

together with

$$\exists G(G \vDash src_{|\varnothing} \wedge G \vDash \neg H) \tag{2}$$

does not imply

$$\exists G(G \vDash src_{|\varnothing} \wedge G \vDash \neg F \wedge G \vDash \neg H). \tag{3}$$

In fact, it is still possible that any graph G with $G \vDash src_{|\varnothing}$ violates either $\neg F$ or $\neg H$ and hence, $\mathcal{F} \wedge \mathcal{H}$. In other words, we may still have $src_{|\varnothing} \vDash \neg(\mathcal{F} \wedge \mathcal{H})$ (equivalent to $F \wedge \mathcal{H} \vDash \neg src_{|\varnothing}$): the composed patterns contradict our counterexample in question, which makes it a false negative.

This case can only occur when non-trivial composed negative application conditions are involved; otherwise, (1) and (2) do indeed imply (3). With higher numbers of guaranteed and forbidden patterns with such non-trivial conditions, the occurrence of false negatives of this class becomes more likely. In order to eliminate these false negatives, we would require to check conditions of the form $src_{|\varnothing} \vDash \neg(\mathcal{F} \wedge \mathcal{H})$ (or equivalently $\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{|\varnothing}$) instead of inferring its result by considering individual patterns ($src_{|\varnothing} \vDash F$, $src_{|\varnothing} \vDash H$) only. There are approaches able to answer these questions [Pen09, SLO18]. Unfortunately, this general problem is undecidable and computationally expensive to solve.

In Chapter 7, Section 7.4, we explain an extension attempting to solve *implication with composed graph patterns* (such as $\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{|\varnothing}$). As a side effect, it also addresses the first class of false negatives. However, it cannot guarantee termination unless forced to terminate when reaching a threshold, e.g. on runtime. As a result, while the number of false negatives will be reduced, some may still occur. In general, the questions of non-termination and incompleteness – false negatives – are inherent to the problem and not specific to our approach. All symbolic approaches allowing a similar degree of expressive power will face the same challenge.

Insufficient information in patterns. False negatives of this class occur when Theorem 6.8 (p. 120) cannot confirm implication of a pattern by a reduced source or target pattern because the implying pattern is missing information.

To illustrate the most common case, consider an s/t-pattern sequence $seq = src_1 \Rightarrow_{b_1} (tar_1, src_2) \Rightarrow_{b_2} tar_2$ created by the Seq-construction. Since the construction propagates information in backward direction only, src_1 and tar_1 (and, by step SC$_\text{k}$-1$^+$, (tar_1, src_2)) contain information about the application of b_1 – tar_2, however, does not. Thus, nodes and edges required for the application of b_1 are not included in the context of tar_2, although they are not necessarily deleted by the rule. Every satisfying transformation sequence $G_0 \Rightarrow_{b_1} G_1 \Rightarrow_{b_2} G_2$ will have G_2 contain these nodes and edges (unless they are deleted) – and, by definition, $G_2 \vDash tar_{2|\varnothing}$. Still, checking the implication $tar_{2|\varnothing} \vDash H$ for a guaranteed pattern H may yield a negative result because of the nodes and edges missing in $tar_{2|\varnothing}$, even if all graphs G_2 in satisfying transformation sequences satisfy H.

In short, analysis of (reduced) source and target patterns to the right may lack information that would be available elsewhere (to the left) in the s/t-pattern sequence. This type of false negatives can be avoided altogether by also propagating information in forward direction. This process of *forward propagation* is described as an extension to our restricted approach in Chapter 7, Section 7.1.

The extension also takes a second (rarer) case of missing information into account. When a rule creates a node, it cannot be connected to other nodes unless the respective edges are created along with it. This is the reverse case of rules deleting nodes: all edges adjacent to deleted nodes have to be explicitly deleted by the rule. While the latter case is encoded in s/t-pattern sequences by shifting Appl(b) to the respective source pattern in steps SC$_1$-3 and SC$_\text{k}$-3, the former case (for target patterns) was not included in the restricted approach because of its complexity in comparison to its low impact.

Often, it makes sense to analyze a set of s/t-pattern sequences constructed by the Seq-construction without considering forward propagation. Forward propagation may come with significant computational effort; sometimes, that effort may not be necessary because counterexamples can be discarded without it. Furthermore, forward propagation is not needed if the purpose of verification is to first find a potential candidate for a k-inductive invariant. Then, verification results are intended to guide the development of an invariant and the process of adjusting system behavior (where appropriate). Even without forward propagation, counterexamples may reveal problems that need to be fixed. If, however, the intention is to verify a k-inductive invariant and if verification without forward propagation results in counterexamples, application of forward propagation may reveal the counterexamples to be spurious.

Accepting the possiblity of false negatives is part of striking a balance between expressive power (in system specifications), termination, and reasonable degrees of performance and completeness. In the evaluation in Chapter 9, we will see that the approach is still capable of verifying meaningful system specifications and application scenarios. In the following chapter, several extensions will allow us to shift the balance of this compromise depending on the requirements of the application scenarios at hand.

This concludes the restricted approach to k-inductive invariant checking. In summary, the restricted formal model, symbolic encoding, and the constructions and theorems in this chapter provide a formalization (**Formal-restricted**) and let us derive an implementation (**Impl.-restricted**) for our verification problem; together, they are the central part of this thesis's contribution.

7. Extensions for the Restricted Approach

The intended advantage of the restricted approach (Chapter 6) in comparison to the general approach (Chapter 5) is its efficiency with respect to computational effort (**Appl.-performance**), which includes guaranteeing termination (**Appl.-termination**). Its biggest drawback is that the approach is not complete (**Appl.-deg.completeness**).

However, this is not a binary choice: in comparison to a hypothetical implementation of the general approach, there are a couple of options for the formalization and implementation of the restricted approach that can improve the approach with respect to completeness – while making concessions in terms of performance. In this chapter, we will discuss these options as extensions to the restricted approach and elaborate on their formalization and implementation.

Outline. This chapter will be structured as follows:

In Section 7.1, we will discuss forward propagation. Since the Seq-construction only accumulates context in backward direction (to the left) during its construction, the analysis of target/source patterns to the right of an s/t-pattern sequence may lack information helpful in discarding sequences as false negatives. While this lack of information does not make the approach unsound, it is one factor leading to imcompleteness. Forward propagation then propagates context in an s/t-pattern sequence from the left to the right and will be shown (in Chapter 9) to reduce the number of false negatives. As such, it will contribute towards **Appl.-deg.completeness**. Forward propagation is implemented as an additional computation step between the construction of sequences and their analysis.

Section 7.2 will focus on rule priorities as a form of more fine-grained control over the application of transformation rules. In essence, rules can be equipped with integers as priorities, with applicability of higher-priority rules preventing the application of rules of lower priority. Taking rule priorities into account requires extending the analysis of s/t-pattern sequences so that sequences with rule applications that violate the priorities are discarded.

Section 7.3 reintroduces partial negative application condition as a means of improving the performance of the approach. In particular, partial negative application condition allow a more efficient handling and evaluation of composed negative application conditions, whose transformation over morphisms is the most costly factor in the approach's implementation. Using partial negative application conditions changes the Seq-construction and the analysis of s/t-pattern sequences.

Section 7.4 addresses a drawback in the restricted approach's analysis of s/t-pattern sequences. Theorems T.2r and T.3r compare a sequence's source and target patterns separately with individual forbidden and guaranteed patterns (in terms of implication), without taking potential interactions between all forbidden or guaranteed patterns into account. This section describes means to consider such interactions by extending the concept of implication of graph patterns introduced earlier. As such, it contributes to **Appl.-deg.completeness** – however, this may come at the cost of higher computational effort or, depending on the approach chosen, non-termination. Including more complex notions of pattern implication requires extending the analysis of s/t-pattern sequencees.

All these sections will follow the same basic structure: after an explanation of the problem, we will address the solution and required changes to the formal model, symbolic encoding, the construction of s/t-pattern sequences (as in Theorem T.1r), and the analysis of s/t-pattern sequences (as in Theorems T.2r, T.3r, T.4r, and T.5r) in separate subsections. Depending on the specific extension, some elements do not require modifications and the respective subsections will be omitted. All sections will close with a subsection dedicated to implementation.

Section 7.5 will offer a formal perspective on applying and combining the extensions introduced in this chapter. Finally, in Section 7.6, we will summarize and discuss the expected effects of the extensions in comparison to the restricted approach.

7.1. Forward Propagation

In the general and restricted approaches, the Seq-construction accumulates context in backward direction by transferring application conditions over reverse rule applications via the L-construction. That or other information is not propagated in forward direction to other target/source and target patterns to the right. That does not make those source and target patterns incorrect, but it means that they describe an over-approximation of the graphs in a satisfying transformation sequence and could be refined.

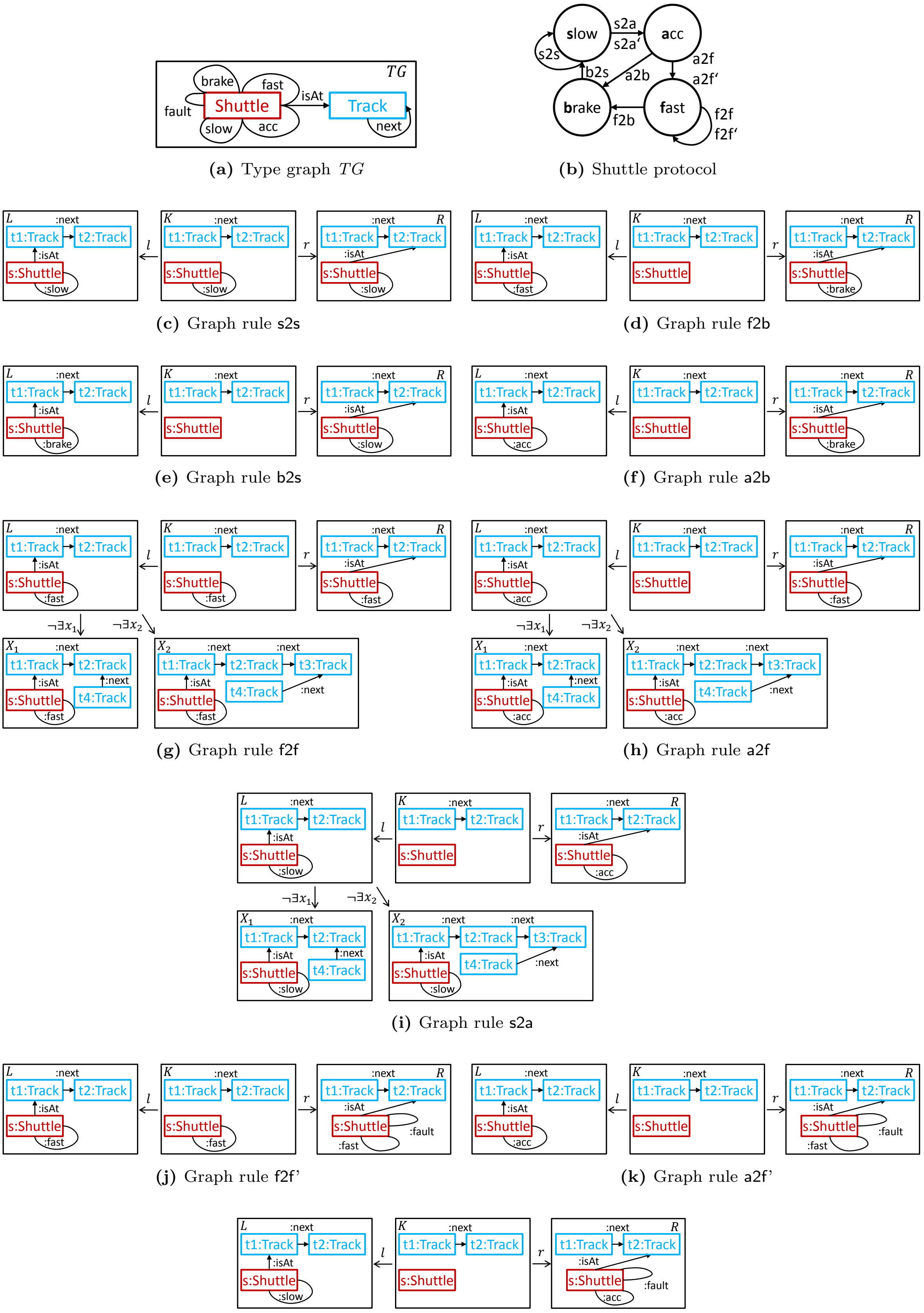
In the general approach, this did not affect the verification result: the target of analysis is always the leftmost source pattern. In the restricted approach, this is not the case: for instance, guaranteed patterns are not shifted to elements in the sequence and hence, not propagated further to the left by the Seq-construction. Instead, we check all patterns in sequences, including intermediate target/source patterns and the rightmost target pattern for implication of guaranteed (and forbidden) patterns during the analysis (Theorems T.2r (p. 143) and T.3r (p. 149)).

Hence, intermediate target/source patterns should ideally have the same information as the leftmost source pattern – otherwise, there may be cases of false negatives due to insufficient information in patterns, as explained in Section 6.8. For example, information about prior rule applications is added to a sequence in each execution of step $\mathrm{SC_k}$-1 of the Seq-construction, which may include nodes and edges that are never deleted – and hence, should be reflected in target/source patterns to the right. While possibly detrimental to performance (**Appl.-performance**), we gain some degree of completeness (**Appl.-deg.completeness**). Depending on the scenario, we may prefer one over the other – and the notion of *forward propagation* described in this section enables us to make a choice.

In order to better explain the problem, we consider the following example system:

Example 7.1 (running example). We extend the shuttle protocol example used in Chapters 5 and 6 by allowing rules to fail the check for switches ahead. Rules s2s, f2b, b2s, and a2b (Figures 7.1(c)-7.1(f)) remain unchanged. Rules f2f, a2f, and s2a (Figures 7.1(g)-7.1(i)) are similar to before; however, they check both the target track and its subsequent track for switches. In addition, rules f2f′, a2f′, and s2a′ are also part of the set of rules $\mathcal{R}$ in our graph transformation system $GTS = (TG, \mathcal{R})$. These three rules do not have any negative application conditions – they are meant to model sensor faults resulting in potential illegal applications of accelerating rules when there is a switch ahead. We keep track of these sensor faults by adding a fault edge to the shuttle on each application. Note that the edge type fault has also been added to the type graph (Figure 7.1(a)).

In this example, we use a composed forbidden pattern $\mathcal{F} = \neg F_1$ that only prohibits a shuttle driving on a switch in speed mode fast (Figure 7.2(a)). The composed guaranteed pattern $\mathcal{H} = \neg \mathcal{H}_1 \wedge ... \wedge \neg H_{16}$ includes all individual guaranteed patterns used before (Example 6.1 (p. 111)) and has another guaranteed pattern H_{16} added to it (Figure 7.2(b)). The (negated) pattern expresses the absence of two sensor faults; in other words, it describes a single fault assumption. This is an example for a guaranteed pattern used to model an external assumption. Obviously, we cannot exclude the occurrence of two sensor faults, but its absence is considered part of the system's specification: safety issues due to two sensor faults are not to be covered by



(a) Type graph TG

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

(f) Graph rule a2b

(g) Graph rule f2f

(h) Graph rule a2f

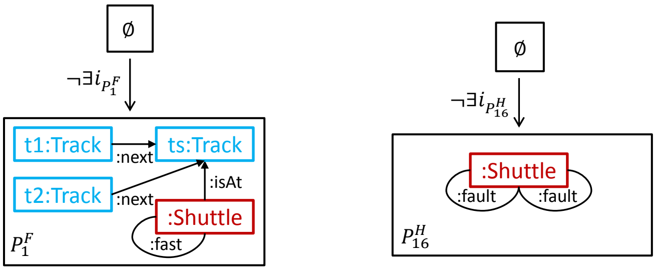
(i) Graph rule s2a

(j) Graph rule f2f'

(k) Graph rule a2f'

(l) Graph rule s2a'

Figure 7.1. – Graph transformation system $GTS = (TG, \mathcal{R})$



(a) Constraint $\neg F_1 = \neg\exists i_{P_1^F}$

(b) Constraint $\neg H_{16} = \neg\exists i_{P_{16}^H}$

Figure 7.2. – Safety property $\mathcal{F} = \neg F_1$ and fragment $\neg H_{16}$ of guaranteed constraint $\mathcal{H} = \neg\mathcal{H}_1 \wedge \ldots \wedge \neg H_{16}$

system verification. One reason for this assumption may be that the probability of two sensor faults occurring is beneath a certain threshold. As before, this approach to system verification may raise ethical and legal questions; however, those are beyond the scope of this thesis.

All elements of this example are also shown in Section C.1.4 of Appendix C.

For the sake of brevity, we focus on the inductive step of our verification approach here, which leaves us with the following verification problem: is $\mathcal{F} = \neg F_1$ a 2-inductive invariant of *GTS* under $\mathcal{H}$, which includes the assumption of at most one rule application occurring without the appropriate checks for switches? △

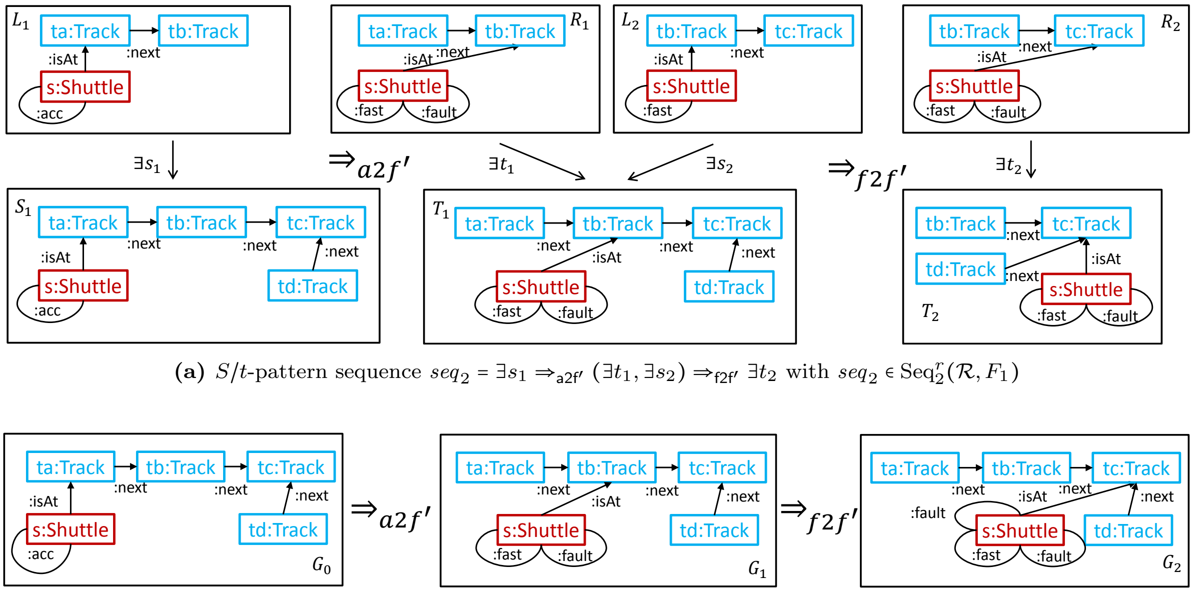
Example 7.2 (insufficient information and incompleteness)**.** In order to verify $\neg F_1$ as a 2-inductive invariant, we will apply Theorems T.1r (p. 130) and T.2r (p. 143), starting with the computation of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$. Without showing all counterexamples here, the result will be negative: $\mathcal{F}$ cannot be delcared a 2-inductive invariant under $\mathcal{H}$. Consider the s/t-pattern sequence $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$ shown in Figure 7.3(a) as one counterexample. It shows a sequence of two rule applications a2f′ and f2f′. Its rightmost target pattern $\exists t_2$ has the forbidden pattern F_1 in its context (T_2) and hence, satisfying transformation sequences will lead to a violation of $\neg F_1$. Furthermore, we cannot detect violations of guaranteed or forbidden patterns earlier in the sequence; in particular, neither reductions of $\exists s_1$ nor of $\exists t_1$ will imply F_1 or H_1 to H_{16}.

However, when we consider satisfying transformation sequences, we will necessarily find a shuttle with two fault edges in the rightmost graphs of all such sequences; Figure 7.3(b) shows an example. This makes sense: two faulty rules (a2f′ and f2f′) have been applied and hence, two fault edges must have been created. The s/t-pattern sequence seq_2 is a false negative, as are all other remaining counterexamples in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$. However, there is insufficient information in the target pattern $\exists t_2$ in order to discard the s/t-pattern sequence during the analysis described by Theorem T.2r (p. 143). In particular, the creation of the fault edge by application of a2f′ (seen in T_1) is not propagated to the right. Thus, T_2 only includes the fault edge created by application of f2f′. △

Addressing this issue requires modification of the construction and anaylsis of s/t-pattern sequences.

7.1.1. Construction of Pattern Sequences

In order to solve the problem sketched above, we apply forward propgation to s/t-pattern sequences created by the Seq-construction (Theorem T.1r (p. 130)). By recursive application of the L-construction to the leftmost source pattern and the inverse rules of the sequence, we transfer context to the next target/source pattern and, subsequently, to all target/source patterns and the rightmost target pattern. Lemma 7.3 justifies that process by establishing that



(a) S/t-pattern sequence $seq_2 = \exists s_1 \Rightarrow_{\mathrm{a2f'}} (\exists t_1, \exists s_2) \Rightarrow_{\mathrm{f2f'}} \exists t_2$ with $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$

(b) Transformation sequence $trans = G_0 \Rightarrow_{\mathrm{a2f'},m_1,m_1'} G_1 \Rightarrow_{\mathrm{f2f'},m_2,m_2'} G_2$ with $trans \models seq$

Figure 7.3. – S/t-pattern sequence with incomplete information and satisfying transformation sequence

the set of represented transformation sequences remains unchanged. What does change is the amount of information available in target and source patterns to the right of the leftmost source and target pattern. The source and target patterns of an s/t-pattern sequence are enriched in order to discard false negatives in the subsequent analysis step (i.e. by Theorem T.2r (p. 143) or Theorem T.3r (p. 149)).

Lemma 7.3 (forward propagation over sequences, modified from the original source [3]). *Given a set of graph rules $\mathcal{R}$, a pattern F, and the set of sequences constructed by* $\mathrm{Seq}_k^r(\mathcal{R}, C)$, *we describe forward propagation as a function* prop *such that for all $seq \in \mathrm{Seq}_k^r(\mathcal{R}, C)$, we have $seq \equiv \mathrm{prop}(seq)$.*

Construction. *We construct* prop *recursively as follows:*

$$\begin{aligned}
&\mathrm{prop}(src_1 \Rightarrow_{b_1} tar_1) = src_1 \Rightarrow_{b_1} tar_1' \\
&\mathrm{prop}(src_1 \Rightarrow_{b_1} tar_1, \qquad\qquad src_2 \Rightarrow_{b_2} tar_2, \ldots, src_k \Rightarrow_{b_k} tar_k) \\
&\quad = src_1 \Rightarrow_{b_1} tar_1', \mathrm{prop}(\mathrm{comb}(tar_1', src_2) \Rightarrow_{b_2} tar_2, \ldots, src_k \Rightarrow_{b_k} tar_k),
\end{aligned}$$

where, given $tar_1 = \exists(t_1 : R_1 \hookrightarrow T_1, ac_{T_1})$,

$$tar_1' = \begin{cases} \mathrm{false} & \textit{if } \mathrm{L}(b_1^{-1}, src_1) = \mathrm{false} \\ \exists(t_1, ac_{T_1'}) & \textit{if } \mathrm{L}(b_1^{-1}, src_1) \textit{ has the form } \exists(t_1, ac_{T_1}') \\ \exists(t_1' \circ t_1, ac_{T_1'}) & \textit{if } \mathrm{L}(b_1^{-1}, src_1) \textit{ has the form } \exists(t_1' \circ t_1, ac_{T_1'}') \end{cases}$$

with

$$ac_{T_1'} = \begin{cases} ac_{T_1}' \wedge \mathrm{Shift}(t_1, \mathrm{Appl}(b_1^{-1})) & \textit{if } \mathrm{L}(b_1^{-1}, src_1) \textit{ has the form } \exists(t_1, ac_{T_1}') \\ ac_{T_1'}' \wedge \mathrm{Shift}(t_1' \circ t_1, \mathrm{Appl}(b_1^{-1})) & \textit{if } \mathrm{L}(b_1^{-1}, src_1) \textit{ has the form } \exists(t_1' \circ t_1, ac_{T_1'}') \end{cases}$$

and, given $src_2 = \exists(s_2' \circ s_2 : L_2 \hookrightarrow T_1, ac_{S_2})$, comb *is defined as follows:*

$$\mathrm{comb}(tar_1', src_2) = \begin{cases} \text{false} & \textit{if } tar_1' = \text{false} \ \textit{ or } src_2 = \text{false} \\ \exists(\qquad s_2' \circ s_2, ac_{T_1'}) & \textit{if } tar_1' = \exists(\qquad t_1, ac_{T_1'}) \\ \exists(t_1' \circ s_2' \circ s_2, ac_{T_1'}) & \textit{if } tar_1' = \exists(t_1' \circ t_1, ac_{T_1'}) \end{cases}$$

$$\begin{tikzcd}
ac_{L_1} \triangleright L_1 \arrow[dd, hook, "s_1"'] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[dd, hook] & R_1 \arrow[dd, hook, "t_1"'] & \varnothing \arrow[l, hook'] \arrow[d, hook] \\
 & & ac_{T_1} \vartriangle & ac_{L_2} \triangleright L_2 \arrow[d, hook, "s_2"'] \\
ac_{S_1} \triangleright S_1 & D_1 \arrow[l, hook'] \arrow[r, hook] & T_1 & S_2 \arrow[l, hook', "s_2'"'] \\
 & & ac_{T_1'} \triangledown & ac_{S_2} \triangledown
\end{tikzcd}
\qquad
\begin{tikzcd}
ac_{L_1} \triangleright L_1 \arrow[dd, hook, "s_1"'] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[dd, hook] & R_1 \arrow[dd, hook, "t_1"'] & \varnothing \arrow[l, hook'] \arrow[d, hook] \\
 & & ac_{T_1} \vartriangle & ac_{L_2} \triangleright L_2 \arrow[d, hook, "s_2"'] \\
ac_{S_1} \triangleright S_1 \arrow[d, hook, "t_0'"'] & D_1 \arrow[l, hook'] \arrow[r, hook] & T_1 \arrow[d, hook, "t_1'"'] & S_2 \arrow[l, hook', "s_2'"'] \\
ac_{T_0'} \triangleright T_0' & & ac_{T_1'} \triangleright T_1' & ac_{S_2} \triangledown
\end{tikzcd}$$

with the second and third case of tar_1' *and* comb *illustrated in the left and right diagram above, respectively. The third cases applies to source patterns (depicted on the right)* $src_1 = \exists(t_0' \circ s_1, ac_{T_0'})$ *with* $\exists s_1 = \mathrm{L}(b_1, \exists t_1)$*; then,* $\mathrm{L}(b_1^{-1}, src_1) = \exists(t_1' \circ t_1, ac_{T_1'})$.

Proof. We will show the required statement by structural induction. We will consider the second case of comb and third case of tar_1' only; the other cases work analogously.

Base case. Let $seq = src_1 \Rightarrow_{b_1} tar_1$ be an s/t-pattern sequence constructed as part of a Seq_k- and prop-construction with $src_1 = \exists(s_1, ac_{S_1})$ and $tar_1 = \exists(t_1, ac_{T_1})$. Then, $\mathrm{prop}(seq) = src_1 \Rightarrow_{b_1} tar_1'$. Furthermore, let $trans = G_0 \Rightarrow_{b_1, m_1, m_1'} G_1$ be a transformation sequence. We need to show $trans \vDash seq \Leftrightarrow trans \vDash \mathrm{prop}(seq)$.

Only if. Assume $trans \vDash seq$. Then, $m_1 \vDash src_1$ and with the L-construction we have $m_1' \vDash L(b_1^{-1}, src_1)$. Assuming $L(b_1^{-1}, src_1) = \exists(t_1' \circ t_1, ac'_{T_1'})$ (third case), we get an injective morphism $x' : T_1' \hookrightarrow G_1$ with $m_1' = x' \circ t_1' \circ t_1$ and $x' \vDash ac'_{T_1'}$. With $m_1' \vDash \mathrm{Appl}(b_1^{-1})$, we get $x' \vDash \mathrm{Shift}(t_1' \circ t_1, \mathrm{Appl}(b_1^{-1}))$, implying $q \vDash tar_1'$. Then, $m_1' \vDash tar_1'$. $m_1 \vDash src_1$ and $m_1' \vDash tar_1'$ imply $trans \vDash \mathrm{prop}(seq)$.

If. Assume $trans \vDash \mathrm{prop}(seq)$. Then, $m_1' \vDash tar_1'$ with $tar_1' = \exists(t_1' \circ t_1, ac_{T_1'})$, i.e. $m_1' \vDash L(b_1^{-1}, src_1)$ and with the L-construction we have $m_1 \vDash src_1$. By construction of $\mathrm{Seq}_k^r(\mathcal{R}, C)$, there is an underlying pattern $src_1^- = L(b_1, tar_1)$ (with $src_1^- = \exists(s_1, ac_{S_1})$ as in the diagram above) such that $m_1 \vDash src_1^-$ and hence, $m_1' \vDash tar_1$, implying $trans \vDash seq$.

$$\begin{tikzcd}
 & & ac_{L_1} \triangleright L_1 \arrow[dd, hook, "s_1"'] \arrow[ddddll, hook, bend right, "m_1 \vDash ac_{L_1}"'] \arrow[dd, phantom, "(1)", shift left=6ex] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[dd, hook] \arrow[dddd, hook, bend left] \arrow[l, phantom, "\triangledown ac_{L_1}"', shift right=4ex] & R_1 \arrow[dd, hook, "t_1"'] \arrow[dddd, hook, "m_1'"] \arrow[dd, phantom, "(2)"', shift right=6ex] & \varnothing \arrow[l, hook', "i_{R_1}"'] \arrow[d, hook, "i_{L_2}"'] \\
 & & & & & L_2 \arrow[d, hook, "s_2"'] \arrow[ddd, hook, bend left=60, "m_2 \vDash \exists(s_2, ac_{S_2}) = src_2"] \\
 & T_0 \arrow[dr, hook] & S_1 \arrow[l, hook'] \arrow[l, phantom, "="', shift right=5ex] & D_1 \arrow[l, hook'] \arrow[r, hook, "r_1^+"'] \arrow[d, hook, "d_1'"'] \arrow[l, phantom, "\vartriangle ac_{S_1}", shift left=3ex] & T_1 \arrow[d, hook, "t_1'"'] \arrow[ddr, hook, "y'"] \arrow[l, phantom, "ac_{T_1} \vartriangle", shift left=3ex] & S_2 \arrow[l, hook', "s_2'"'] \arrow[dd, hook, "y"] \arrow[l, phantom, "ac_{S_2} \vartriangle", shift left=3ex] \\
 & & ac_{T_0'} \triangleright T_0' \arrow[dll, hook] & D_1' \arrow[l, hook'] \arrow[r, hook, "r_1'"] \arrow[d, hook, "k_1'"'] \arrow[l, phantom, "(3)", shift left=5ex] \arrow[r, phantom, "(4)", shift left=5ex] & T_1' \arrow[dr, hook, "x'"] & \\
G_0 & & & K_1' \arrow[lll, hook'] \arrow[rr, hook, "r_1^*"] \arrow[l, phantom, "(5)", shift left=5ex] \arrow[r, phantom, "ac_{T_1'} \triangledown \quad (6)", shift right=5ex] & & G_1
\end{tikzcd}$$

Inductive step. Let $seq' = src_1 \Rightarrow_{b_1} tar_1, src_2 \Rightarrow_{b_2} \ldots \Rightarrow_{b_{k+1}} tar_{k+1}$ be a k+1-sequence of s/t-patterns constructed as part of a Seq_k-construction. Given a transformation sequence $trans' =$

$G_0 \Rightarrow_{b_1,m_1,m_1'} G_1 \Rightarrow_{b_2,m_2,m_2'} \cdots \Rightarrow_{b_{k+1},m_{k+1},m_{k+1}'} G_{k+1}$, we need to show $\mathit{trans}' \vDash \mathit{seq}' \Leftrightarrow \mathit{trans}' \vDash \mathrm{prop}(\mathit{seq}')$.

Only if. Assume $\mathit{trans}' \vDash \mathit{seq}'$. Then, we have $m_1 \vDash \mathit{src}_1$, which implies $m_1' \vDash L(b_1^{-1}, \mathit{src}_1)$ and $m_1' \vDash \mathit{tar}_1'$ with $\mathit{tar}_1' = \exists(t_1' \circ t_1, \mathit{ac}_{T_1'})$ and pushouts (3) and (4).

Furthermore, $\mathit{trans}' \vDash \mathit{seq}'$ implies $(m_1', m_2) \vDash (\mathit{tar}_1, \mathit{src}_2)$ (where $\mathit{tar}_1 = \exists(t_1, \mathit{ac}_{T_1})$ and $\mathit{src}_2 = \exists(s_2' \circ s_2, \mathit{ac}_{T_1}))$. Thus, there exists an injective morphism $y' : T_1 \hookrightarrow G_1$ such that $y' \circ t_1 = m_1'$, $y' \circ s_2' \circ s_2 = m_2$, and $y' \vDash \mathit{ac}_{T_1}$.

By pushout decomposition, (G_1, r_1^*, y') is a pushout and hence, $y' \circ r_1^+ = r_1^* \circ k_1' \circ d_1'$. Since (4) is also a pushout, there is an injective morphism $x' : T_1' \hookrightarrow G_1$ such that $x' \circ r_1' = r_1^* \circ k_1'$ and $x' \circ t_1' = y'$. Then, by pushout decomposition, (G_1, r_1^*, x') is a pushout (6) over r_1' and k_1', implying $x' \vDash \mathit{ac}_{T_1'}$. Furthermore, we have $x' \circ t_1' \circ s_2' \circ s_2 = y' \circ s_2' \circ s_2 = m_2$, i.e. $m_2 \vDash \mathrm{comb}(\mathit{tar}_1', \mathit{src}_2)$. With $x' \circ t_1' \circ t_1 = m_1'$, we get $(m_1', m_2) \vDash (\mathit{tar}_1', \mathrm{comb}(\mathit{tar}_1', \mathit{src}_2))$ and, by inductive hypothesis, $\mathit{trans}' \vDash \mathrm{prop}(\mathit{seq}')$.

If. Assume $\mathit{trans}' \vDash \mathrm{prop}(\mathit{seq}')$. Then, we have $m_1 \vDash \mathit{src}_1$ and, for the respective target/source pattern, $(m_1', m_2) \vDash (\mathit{tar}_1', \mathrm{comb}(\mathit{tar}_1', \mathit{src}_2))$, i.e. there exists an injective morphism $x' : T_1' \hookrightarrow G_1$ such that $x' \vDash \mathit{ac}_{T_1'}$, $x' \circ t_1' \circ t_1 = m_1'$, $x' \circ t_1' \circ s_2' \circ s_2 = m_2$, and (3) and (4) are pushouts. There also exists an injective morphism $y' : T_1 \hookrightarrow G_1$ with $y' = x' \circ t_1'$ such that $y' \circ t_1 = m_1'$ and $y' \circ s_2' \circ s_2 = m_2$. Since $m_1 \vDash \mathit{src}_1$ and by construction of $\mathrm{Seq}_k^r(\mathcal{R}, C)$, there is an injective morphism $q : S_1 \hookrightarrow G_0$ such that $q \vDash \mathit{ac}_{S_1}$. Since (4) + (6) and (3) + (5) are pushouts and by the L-construction, we get $y' \vDash \mathit{ac}_{T_1}$ and hence, $(m_1', m_2) \vDash (\mathit{tar}_1, \mathit{src}_2)$. With the inductive hypothesis, we have $\mathit{trans}' \vDash \mathit{seq}'$. □

Informally, the complex prop-construction can be reduced to the following explanation: first, context is propagated from a source to the next target pattern via the L-construction and the inverse rule. Then, the applicability condition of the inverse version of the subsequent rule it added to the target pattern. If we have not reached the rightmost target pattern, it will be part of a target/source pattern. Since the existential condition of target/source patterns has to have the same codomain, the source pattern has to be extended by applying comb. Then, application of prop uses the extended source pattern to continue the recursion. The extension by comb is the most complex part of the construction and relies on the results of the previous Seq-construction in order to propertly extend and connect the target/source patterns. Still, prop always terminates and yields a finite result by construction.

The distinction between the second and third case of computing tar_1 and comb is necessary because an s/t-pattern sequence's leftmost source pattern is different from source patterns in a target/source pattern of the sequence (if the sequence was created by the Seq-construction). The positive context $s : L_1 \to S_1$ of such a source pattern $\mathit{src}_1 = \exists(s_1, \dots)$ was created from a corresponding target pattern $\mathit{tar}_1 = \exists(t_1, \dots)$ by $\mathrm{L}(b_1, \exists(t_1, \dots)) = \exists(s_1, \dots)$. Hence, application of $\mathrm{L}(b_1^{-1}, \exists(s_1, \dots))$ will again result in $\exists(t_1, \dots)$ (although the composed negative application condition may have changed). This is covered by the second case.

If, however, L is applied to an intermediate source pattern src_{i+1}, the situation is different. Firstly, src_{i+1} is part of a target/source pattern. Because of the combination of target and source patterns in step $\mathrm{SC_k}$-1^+, src_{i+1} will have the structure $\exists(s_i' \circ s_{i+1}, \dots)$ with $\exists(s_{i+1}, \dots) = \mathrm{L}(b_{i+1}, \mathit{tar}_{i+1})$ the result of the previous step $\mathrm{SC_k}$-2 of the Seq-construction (and $\mathit{tar}_{i+1} = \exists(t_{i+1}, \dots))$. As a result, we have $\mathrm{L}(b_{i+1}^{-1}, \exists(s_i' \circ s_{i+1}, \dots)) = \exists(t_{i+1}' \circ t_{i+1}, \dots)$. Secondly, src_{i+1} may already have had context propagated from earlier patterns in the sequence. This leads to $\mathit{src}_{i+1} = \exists(t_i' \circ s_i' \circ s_{i+1}, \dots)$, with similar results for application of L. Both cases are combined as $\mathit{src}_{i+1} = \exists t_i' \circ s_{i+1}$ in the third case of prop and comb. For instance, in the respective diagram on the right, we have $\mathit{src}_1 = \exists(t_0' \circ s_1, \dots)$.

Example 7.4 (forward propagation)**.** Consider Figure 7.4, which depicts the s/t-pattern sequence from Example 7.2 (p. 172), Figure 7.3(a): $\mathit{seq}_2 = \exists \mathit{src}_1 \Rightarrow_{\mathsf{a2f'}} (\mathit{tar}_1, \mathit{src}_2) \Rightarrow_{\mathsf{f2f'}} \mathit{tar}_2$ or,

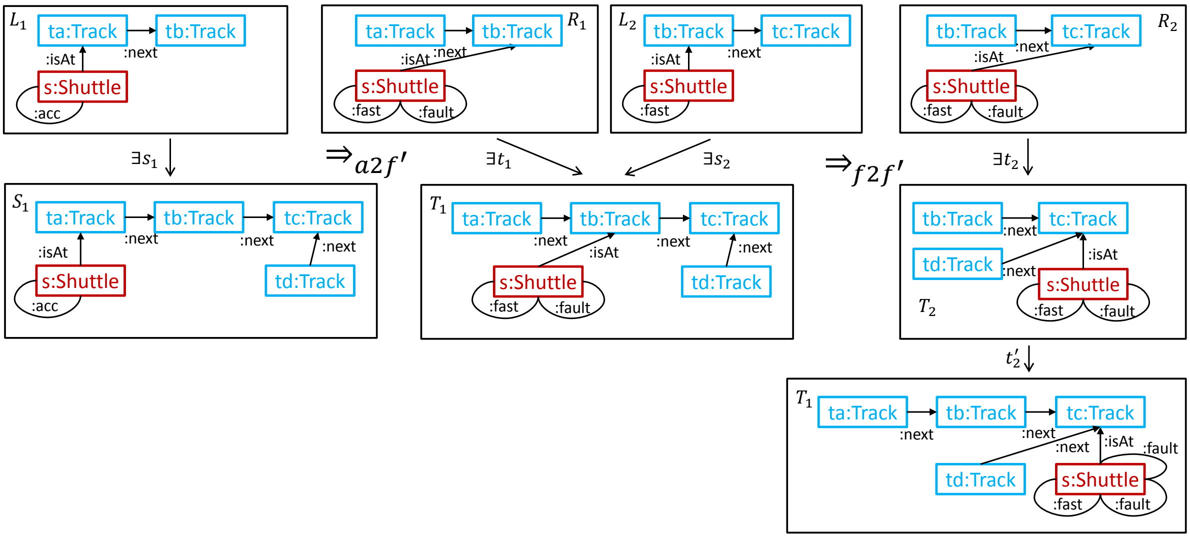


Figure 7.4. – S/t-pattern sequence $seq_2' = \text{prop}(seq_2) = \exists s_1 \Rightarrow_{\mathsf{a2f'}} (\exists t_1, \exists s_2) \Rightarrow_{\mathsf{f2f'}} \exists(t_2' \circ t_2)$ with $seq_2 \in \text{Seq}_2^r(\mathcal{R}, F_1)$

in more detail, $seq_2 = \exists s_1 \Rightarrow_{\mathsf{a2f'}} (\exists t_1, s_2) \Rightarrow_{\mathsf{f2f'}} tar_2$. As mentioned before, $\exists t_2$ does not contain the information about the second sensor fault; furthermore, it does not show the existence of a fourth track ta, although any satisfying transformation sequences will contain both the sensor faults and the track.

Since $\exists s_1$ is the direct result of $\text{L}(\mathsf{a2f'}, \exists t_1)$ and does not have a non-trivial nested application condition, there is nothing to propagate from $\exists s_1$ to $\exists t_1$ in this example. Formally, we have $\text{L}(\mathsf{a2f'}^{-1}, \exists s_1) = \exists t_1$, which triggers the second case of prop: $tar_1' = \exists t_1$. Consequently, the target/source pattern $(tar_1', src_2) = (tar_1, src_2) = (\exists t_1, \exists s_2)$ is also left unchanged. In particular, we have:

$$\begin{aligned}
\text{prop}(seq_2) &= \text{prop}(src_1 \Rightarrow_{\mathsf{a2f'}} tar_1, src_2 \Rightarrow_{\mathsf{f2f'}} tar_2) \\
&= src_1 \Rightarrow_{\mathsf{a2f'}} tar_1', \text{prop}(\text{comb}(tar_1', src_2) \Rightarrow_{\mathsf{a2f'}} tar_2) \quad (\text{where } tar_1' = \text{L}(\mathsf{f2f'}^{-1}, src_1)) \\
&= src_1 \Rightarrow_{\mathsf{a2f'}} tar_1, \text{prop}(src_2 \Rightarrow_{\mathsf{f2f'}} tar_2)
\end{aligned}$$

Then, $\text{prop}(src_2 \Rightarrow tar_2)$ requires transfer of $\exists s_2$ over the inverse rule of $\mathsf{f2f'}$ to the new target pattern tar_2'. Here, we have $\text{L}(\mathsf{f2f'}^{-1}, \exists s_2) = \exists(t_2' \circ t_2)$ (see Figure 7.4). This is the third case of prop. In particular,

$$\begin{aligned}
\text{prop}(src_2 \Rightarrow_{\mathsf{f2f'}} tar_2) &= src_2 \Rightarrow_{\mathsf{f2f'}} tar_2' \qquad (\text{where } tar_2' = \text{L}(\mathsf{f2f'}^{-1}, src_2)) \\
&= src_2 \Rightarrow_{\mathsf{f2f'}} \exists(t_2' \circ t_2).
\end{aligned}$$

Formally, tar_2' is indeed a new target pattern. In practice and the implementation, however, we can simply add new elements to the context graph T_2 – here, those are the track ta, its connection to tb, and the second fault edge.

Since both rules do not create any elements, their inverse rules do not delete any elements. Thus, both have a trivial applicability condition (i.e. true) – and their transfer (see construction of prop) does not change the respective patterns.

After applying forward propagation, the reduction of $\exists(t_2' \circ t_2)$ to the pattern $\exists(i_{T_2'} : \varnothing \hookrightarrow T_2')$ will indeed imply the guaranteed pattern H_{16} because T_2' contains a shuttle with two fault edges. We would not have found this implication in $\exists(i_{T_2} : \varnothing \hookrightarrow T_2)$. △

7.1.2. Analysis of Pattern Sequences

In general, with longer sequences and with more complex composed negative application conditions in particular, forward propagation can affect the analysis of sequences in the following ways:

- Additional (propagated) nodes and edges may reveal a violation of a different forbidden pattern or a guaranteed pattern in an intermediate target/source pattern or the rightmost target pattern.
- Propagated composed negative application conditions may cause a previously failed implication check to succeed because the propagated condition corresponds to a forbidden or guaranteed pattern's composed negative application condition.

Note that these effects and forward propagation in itself does not change violation or satisfaction of forbidden or guaranteed patterns by graphs of a satisfying transformation sequence. Lemma 7.3 (p. 173) states that for any s/t-pattern sequence constructed by a Seq-construction, its set of represented – i.e., satisfying – transformation sequences is equal to the set of satisfying transformation sequences for the s/t-pattern sequence after forward propagation. Forward propagation does not change satisfiability and satisfying transformation sequences – it simply reveals information that, for the purpose of analyzing individual and intermediate source and target patterns, is hidden in other source or target patterns.

Including forward propagation during the analysis requires changing Theorem T.2r as follows: instead of all sequences $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \mathcal{F})$, we have to consider all sequences $\mathrm{prop}(seq)$ for sequences $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \mathcal{F})$. Conditions (1) and (2) remain the same.

Theorem T.2e-fp (k-inductive invariant checking with forward propagation [3])**.** *Let* $GTS = (\mathcal{R}, TG)$ *be a graph transformation system and* $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$ *and* $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$ *be a composed forbidden pattern and composed guaranteed pattern, respectively.*

$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ if, for all sequences $\mathrm{prop}(seq)$ *with* $\mathrm{prop}(seq) = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ *and* $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$*, one of the following conditions holds:*

1. $\exists z, v(1 \le z \le k \wedge (src_{z|\varnothing} \vDash H_v \vee src_{z|\varnothing} \vDash F_v))$.
2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.

Proof. This follows from the proof of Theorem T.2r (p. 143) and the equivalence $\mathrm{prop}(seq) \equiv seq$ given $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$ (Lemma 7.3 (p. 173)). □

Example 7.5 (forward propagation and 2-inductive invariant checking)**.** Given $seq = \exists s_1 \Rightarrow_{\mathsf{a2f'}} (\exists t_1, \exists s_2) \Rightarrow_{\mathsf{f2f'}} \exists t_2$ as in Example 7.2 (p. 172), Figure 7.3(a), analysis by Theorem T.2r (p. 143) fails to show that seq is a false negative. When we consider Theorem T.2e-fp, we investigate $seq' = \mathrm{prop}(seq) = \exists s_1 \Rightarrow_{\mathsf{a2f'}} (\exists t_1, \exists s_2) \Rightarrow_{\mathsf{f2f'}} \exists t_2'$ (Example 7.4, Figure 7.4 (p. 176)) instead. Then, as mentioned in Example 7.4 (p. 175), we have $\exists i_{T_2'} \vDash \mathcal{H}_{16}$, which is covered by condition (2) of the updated theorem. The s/t-pattern sequence – and, indeed, all other remaining sequences – can be discarded as a counterexample: $\mathcal{F}$ is a 2-inductive invariant of GTS under $\mathcal{H}$. △

Theorem T.3r (p. 149) (and Theorems T.4r (p. 153) and T.5r (p. 155)) can be updated in a similar fashion:

Theorem T.3e-fp (k–1-bounded backward model checking)**.** *Let* $GTS = (\mathcal{R}, TG)$ *be a graph transformation system and* $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$*,* $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$*, and* $\mathcal{S} = \bigwedge_{o\in O} \neg S_o$ *be a composed forbidden pattern, composed guaranteed pattern, and composed start configuration pattern, respectively, with* $\mathcal{S} \vDash \mathcal{F}$*.*

For all graphs $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ *and graph grammars* $GG = (GTS, G_0)$ *with* $GG \in \mathrm{IND}(GTS, \mathcal{S})$*, we have* $G \models \mathcal{F}$*, if for all sequences* $\mathrm{prop}(seq)$ *with* $\mathrm{prop}(seq) = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n$ *and* $seq \in \mathrm{SEQ}^r_{k-1}(\mathcal{R}, \neg\mathcal{F})$*, one of the following conditions holds:*

1. $\exists z, v(1 \leq z \leq n \wedge (src_{z|\varnothing} \models H_v))$.
2. $\exists v(tar_{k|\varnothing} \models H_v)$.
3. $\exists v(src_{1|\varnothing} \models SC_v)$.

Proof. Similar to the proof of Theorem T.2e-fp (p. 177), this theorem can be shown by combining the proof of Theorem T.3r (p. 149) with $\mathrm{prop}(seq) \equiv seq$ given $seq \in \mathrm{Seq}^r_k(\mathcal{R}, \neg\mathcal{F})$ (Lemma 7.3 (p. 173)). □

7.1.3. Implementation

Implementing forward propagation does not require any changes to the functions createSequences or extendSequences. However, we need to add forward propagation after constructing s/t-pattern sequences and make sure that propagated sequences are properly analyzed. Algorithm 7.1 shows an updated version of Algorithm 6.7 (p. 158): after applying extendSequences, all s/t-pattern sequences seq have forward propagation ($\mathrm{prop}(seq)$, described in Algorithm 7.2) applied to them. Each sequence is also the target of another analysis step: although analysis is interwoven in extendSequences (see Algorithm 6.9 (p. 160)), it has to be repeated after applying forward propagation.

Algorithm 7.1: k-invcheck-prop.

```
desc.  : verification scheme for $k$-inductive invariant checking with forward
         propagation (Theorem T.2e-fp)
input  : an integer $k$ with $k \geq 1$, a set $\mathcal{R}$ of graph rules, a set $\mathcal{F}$ of forbidden
         patterns, a set $\mathcal{H}$ of guaranteed patterns
output: a set of $k$-sequences of $s/t$-patterns as counterexamples

$results \leftarrow \varnothing$
foreach $F \in \mathcal{F}$ do                                              /* $\mathrm{Seq}^r_1(\mathcal{R}, F)$ */
    $results \leftarrow results \cup \mathrm{createSequences}(\mathcal{R}, F, \mathcal{H}, \mathcal{F} \cup \mathcal{H})$
for $i \leftarrow 2$ to $k$ do
    $temp \leftarrow \mathrm{extendSequences}(\mathcal{R}, results, \mathcal{F} \cup \mathcal{H})$           /* $\mathrm{Seq}^r_k(\mathcal{R}, F)$ */
    $results \leftarrow \varnothing$
    for $seq \in temp$ do
        $seq' \leftarrow \mathrm{prop}(seq)$                                       /* $\mathrm{prop}(seq)$ */
        if not $\mathrm{discardSequence}(seq', \mathcal{F}, \mathcal{H})$ then                  /* analysis */
            $results \leftarrow results \cup \{seq'\}$
return $results$
```

Algorithm 7.3 shows a similarly modified variant of Algorithm 6.10 (p. 161) for $k-1$-bounded backward model checking. As before, s/t-pattern sequences have to be analyzed again after forward propagation has been applied. Here, this means comparing the sequence's reduced patterns with the guaranteed patterns.

Algorithm 7.2: prop(seq)

desc. : forward propagation for s/t-pattern sequences (Lemma 7.3 (p. 173))

input : an s/t-pattern sequence $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ created by the Seq-construction

output: the s/t-pattern sequence after application of forward propagation

$$\begin{tikzcd}
ac_{L_1} \triangleright L_1 \arrow[dd, hook, "s_1"'] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[dd, hook] & R_1 \arrow[dd, hook, "t_1"] & \varnothing \arrow[l, hook'] \arrow[d, hook] \\
 & & ac_{T_1} \triangle & ac_{L_2} \triangleright L_2 \arrow[d, hook, "s_2"'] \\
ac_{S_1} \triangleright S_1 & D_1 \arrow[l, hook'] \arrow[r, hook] & T_1 & S_2 \arrow[l, hook', "s'_2"'] \\
 & ac_{T'_1} \triangledown & & ac_{S_2} \triangledown
\end{tikzcd}
\qquad
\begin{tikzcd}
ac_{L_1} \triangleright L_1 \arrow[dd, hook, "s_1"'] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[dd, hook] & R_1 \arrow[dd, hook, "t_1"] & \varnothing \arrow[l, hook'] \arrow[d, hook] \\
 & & ac_{T_1} \triangle & ac_{L_2} \triangleright L_2 \arrow[d, hook, "s_2"'] \\
ac_{S_1} \triangleright S_1 \arrow[d, hook, "t'_0"'] & D_1 \arrow[l, hook'] \arrow[r, hook] & T_1 \arrow[d, hook, "t'_1"'] & S_2 \arrow[l, hook', "s'_2"'] \\
ac_{T'_0} \triangleright T'_0 & & ac_{T'_1} \triangleright T'_1 & ac_{S_2} \triangledown
\end{tikzcd}$$

```
if $seq$ has the form $src_1 \Rightarrow_{b_1} tar_1$ then    /* prop, case 1, end of recursion */
    if $\mathrm{L}(b_1^{-1}, src_1) = \mathrm{false}$ then
        $tar'_1 \leftarrow \mathrm{false}$
    else if $\mathrm{L}(b_1^{-1}, src_1) = \exists(t_1, ac'_{T_1})$ then    /* left diagram */
        $tar'_1 \leftarrow \exists(t_1, ac_{T'_1} \wedge \mathrm{Shift}(t_1, \mathrm{Appl}(b_1^{-1})))$
    else if $\mathrm{L}(b_1^{-1}, src_1) = \exists(t'_1 \circ t_1, ac'_{T'_1})$ then    /* right diagram */
        $tar'_1 \leftarrow \exists(t'_1 \circ t_1, ac_{T'_1} \wedge \mathrm{Shift}(t'_1 \circ t_1, \mathrm{Appl}(b_1^{-1})))$
    $seq \leftarrow src_1 \Rightarrow_{b_1} tar'_1$
    return $seq$
else    /* prop, case 2, with $src_2 = \exists(s'_2 \circ s_2)$ */
    if $\mathrm{L}(b_1^{-1}, src_1) = \mathrm{false}$ then
        $tar'_1 \leftarrow \mathrm{false}$
        $src_2 \leftarrow \mathrm{false}$    /* comb, case 1 */
    else if $\mathrm{L}(b_1^{-1}, src_1) = \exists(t_1, ac'_{T_1})$ then    /* left diagram */
        $tar'_1 \leftarrow \exists(t_1, ac_{T'_1} \wedge \mathrm{Shift}(t_1, \mathrm{Appl}(b_1^{-1})))$
        $src_2 \leftarrow \exists(s'_2 \circ s_2, ac_{T'_1} \wedge \mathrm{Shift}(t_1, \mathrm{Appl}(b_1^{-1})))$ (    /* comb, case 2 */)
    else if $\mathrm{L}(b_1^{-1}, src_1) = \exists(t'_1 \circ t_1, ac'_{T'_1})$ then    /* right diagram */
        $tar'_1 \leftarrow \exists(t'_1 \circ t_1, ac_{T'_1} \wedge \mathrm{Shift}(t'_1 \circ t_1, \mathrm{Appl}(b_1^{-1})))$
        $src_2 \leftarrow \exists(t'_1 \circ s'_2 \circ s_2, ac_{T'_1} \wedge \mathrm{Shift}(t'_1 \circ t_1, \mathrm{Appl}(b_1^{-1})))$ (    /* comb, case 3 */)
    $seq \leftarrow src_1 \Rightarrow_{b_1} tar'_1, \mathrm{prop}(src'_2 \Rightarrow_{b_2} \dots \Rightarrow_{b_k} tar_k)$    /* recursion */
    return $seq$
```

Algorithm 7.3: k–1-modelcheck-prop.

```
desc.  : verification scheme for $k$–1-bounded backward model checking using forward
         propagation (Theorem T.3e-fp (p. 177))
input  : an integer $k$ with $k \geq 2$, a set $\mathcal{R}$ of graph rules, a set $\mathcal{F}$ of forbidden patterns,
         a set $\mathcal{H}$ of guaranteed patterns, a set $\mathcal{S}$ of start configuration patterns
output: a set of $k$-sequences of $s/t$-patterns as counterexamples

$sequences \leftarrow \varnothing$
foreach $F \in \mathcal{F}$ do                                                    /* $\mathrm{Seq}^r_1(\mathcal{R}, F)$ */
    $sequences \leftarrow sequences \cup \mathrm{createSequences}(\mathcal{R}, F, \mathcal{H}, \mathcal{H})$
    foreach $seq \in sequences$ with $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n$ do      /* analysis */
        if not $\mathrm{discardPattern}(src_{1|\varnothing}, \mathcal{S})$ then
            $results \leftarrow results \cup \{seq\}$
for $i \leftarrow 2$ to $k - 1$ do
    $temp \leftarrow \mathrm{extendSequences}(\mathcal{R}, sequences, \mathcal{H})$                          /* $\mathrm{Seq}^r_{k-1}(\mathcal{R}, F)$ */
    $sequences \leftarrow \varnothing$
    for $seq \in temp$ do
        $seq' \leftarrow \mathrm{prop}(seq)$                                                /* $\mathrm{prop}(seq)$ */
        if not $\mathrm{discardSequence}(seq', \varnothing, \mathcal{H})$ then                        /* analysis */
            $sequences \leftarrow sequences \cup \{seq'\}$
    foreach $seq \in sequences$ with $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n$ do      /* analysis */
        if not $\mathrm{discardPattern}(src_{1|\varnothing}, \mathcal{S})$ then
            $results \leftarrow results \cup \{seq\}$
return $results$
```

7.2. Rule Priorities

One extension of the restricted formal model has been described and discussed in earlier work [BBG+06, Dyc12] for 1-inductive invariant checking: rule priorities. Rule priorities are a lesser form of control programs, which are an established concept [Pen09, GdMR+12] to steer rule applicability and application beyond nested application conditions.

Rule priorities work as follows: when matches for multiple rules can be found in a specific graph – and those matches also satisfy the respective application conditions – only those (applicable) rules with the highest priority may be applied. In this section, we discuss an extension that modifies the restricted approach to take rule priorities into account. In terms of this thesis's contribution, it contributes mainly to the approach's applicability to different classes of examples. Supporting rule priorities requires reiterating the respective definitions in the context of the restricted formal model. While the construction of sequences does not need to be changed, the theorem for their analysis will be updated.

7.2.1. Formal Model

Priorities are formalized by a function mapping rules to natural numbers [BBG+06, BG08b] as defined in the following definition, which refines Definition 2.20 (p. 32).

Definition 7.6 (graph rules with priorities, graph transformation (sequences) with priorities)**.** *Given a set of graph rules $\mathcal{R}$, rule priorities are defined as a function $prio : \mathcal{R} \to \mathbb{N}$. The application of a graph rule with priorities $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$ with $b \in \mathcal{R}$ consists of two pushouts* (1) *and* (2) *such that $G_0 \Rightarrow_{b,m,m'} G_1$ is a graph transformation and there does not exist a rule $b' \in \mathcal{R}$ such that $prio(b') > prio(b)$ and there is a transformation $G_0 \Rightarrow_{b',n,n'} G_1'$.*

$$\begin{tikzcd}
ac_L \triangleright L \arrow[d, hook, "m"'] & K \arrow[l, hook', "l"] \arrow[r, hook, "r"'] \arrow[d] \arrow[dl, phantom, "(1)"] \arrow[dr, phantom, "(2)"] & R \triangleleft ac_R \arrow[d, hook, "m'"] \\
G_0 & D \arrow[l, hook'] \arrow[r, hook] & G_1
\end{tikzcd}$$

We denote the transformation with priorities as $G_0 \Rightarrow^{prio}_{b,m,m'} G_1$. We write $G_0 \Rightarrow^{prio}_{b} G_1$ or $G_0 \Rightarrow^{prio} G_1$ to express that there exist m, m' (or b, m, and m', respectively) such that $G_0 \Rightarrow^{prio}_{b,m',m'} G_1$. We also write $G_0 \Rightarrow^{prio}_{\mathcal{R}} G_1$ to express that there exists a rule $b \in \mathcal{R}$ such that $G_0 \Rightarrow^{prio}_{b} G_1$.

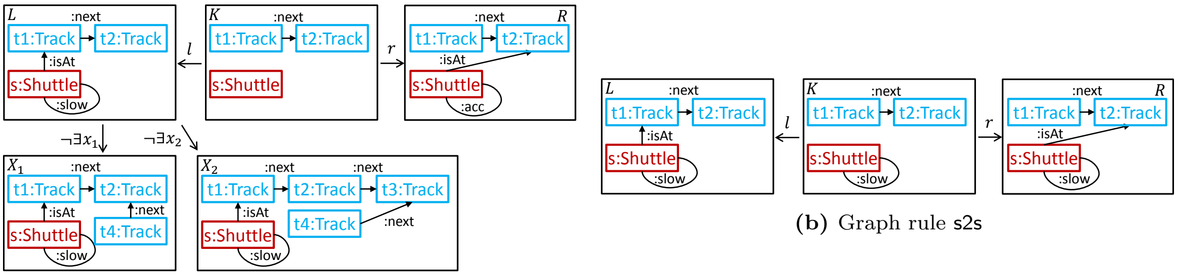
Similarly, for transformation sequences, $G_0 \Rightarrow^{prio}_{\mathcal{R}} G_1 \Rightarrow^{prio}_{\mathcal{R}} \ldots \Rightarrow^{prio}_{\mathcal{R}} G_k$ expresses subsequent graph transformations $G_0 \Rightarrow^{prio}_{\mathcal{R}} G_1, \ldots, G_{k-1} \Rightarrow^{prio}_{\mathcal{R}} G_k$ and is also denoted by $G_0 \Rightarrow^{k,prio}_{\mathcal{R}} G_k$ or, where specific rules, matches or comatches are of interest, by $G_0 \Rightarrow^{prio}_{b_1,m_1,m_1'} \ldots \Rightarrow^{prio}_{b_k,m_k,m_k'} G_k$.

Intuitively, graph rules with priorities may only be applied if there is no rule of higher priority ($prio(b') > prio(b)$) applicable. Note that, since the existence of other applicable rules of higher priority needs to be considered, the concept of graph transformations with priorities always relates to a set of rules over which priorities are defined.

Similar to Definitions 2.20 and 7.6 (p. 181), we refine Definition 2.24 (p. 34) as follows:

Definition 7.7 (typed graph transformation system with priorities)**.** *A* typed graph transformation system with priorities *$GTS = (TG, \mathcal{R}, prio)$ consists of a type graph TG, a set of typed graph rules $\mathcal{R}$, and a priority function $prio : \mathcal{R} \to \mathbb{N}$.*

Since the addition of rule priorities changes the notion of rule applicability, we also need to refine the concepts of graph grammars and their state spaces:



(a) Graph rule s2a

(b) Graph rule s2s

Figure 7.5. – Graph rules from running examples without priorities

Definition 7.8 (typed graph grammar with priorities, state space)**.** *A* typed graph grammar with priorities *$GG = (G_0, GTS)$ consists of an initial graph G_0 and a typed graph transformation system with priorities $GTS = (TG, \mathcal{R}, prio)$.*

The state space *of a graph grammar $GG = ((TG, \mathcal{R}, prio), G_0)$ is defined as* $\text{REACH}(GG) = \{G \mid \exists n(G_0 \Rightarrow_{\mathcal{R}}^{n,prio} G)\}$*, i.e. as the set of graphs reachable by graph transformations from the initial graph, taking priorities into account.*

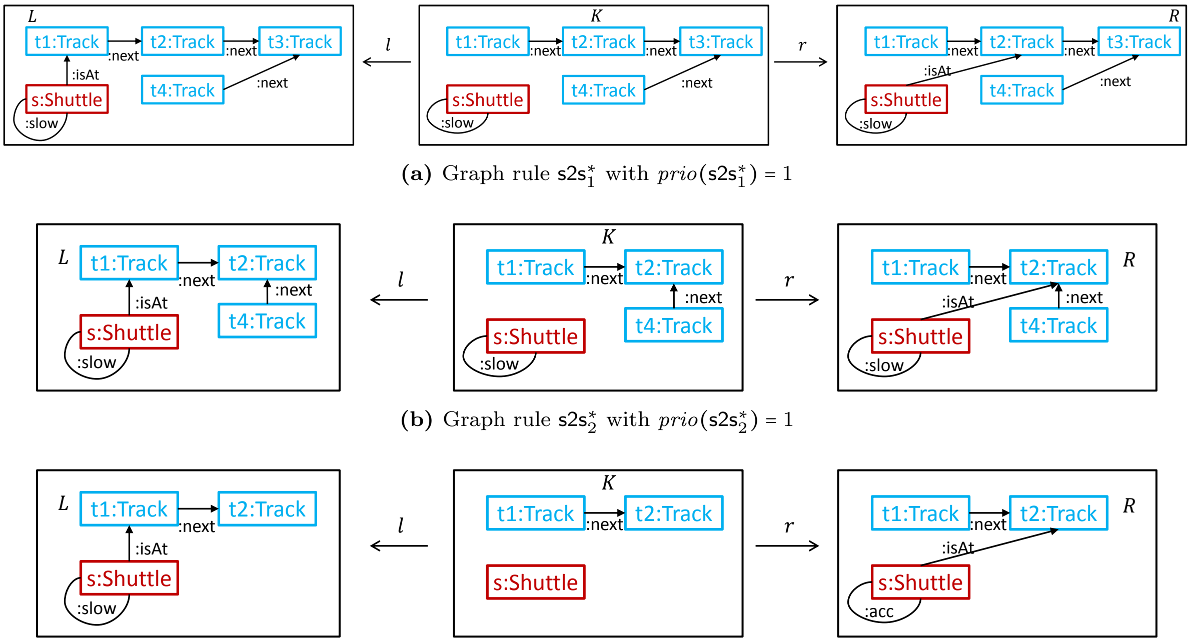
A common application of rules with priorities is to make sure that the application of certain rules can be preempted by higher-priority rules, where necessary – for instance, to prevent erroneous behavior or to repair an erroneous or dangerous situation as soon as it occurs.

Example 7.9 (graph rules with priorities, running example)**.** Figure 7.5(a) shows the graph rule s2a (seen, for example, in Example 6.1 (p. 111)) with a composed negative application condition: the shuttle is only allowed to accelerate if there does not exist a switch one or two tracks ahead. Thus, whenever the system finds a match of the left rule side L as a subgraph of the situation described by X_1 or X_2, only the rule s2s (Figure 7.5(b)) will be applicable, keeping the speed mode unchanged.

This is a precaution aiming at preventing a violation of the safety property – a fast shuttle on a switch. Using priorities, we can achieve a similar effect without the composed negative application condition $\neg\exists x_1 \wedge \neg\exists x_2$. Figure 7.6(a) shows a graph rule $\mathsf{s2s}_1^*$ with a priority of 1, i.e. $prio(\mathsf{s2s}_1^*) = 1$. Its left side and the context of the negative application condition $\neg\exists x_2$ in rule s2a above are isomorphic. Hence, whenever $\neg\exists x_2$ would prevent the application of s2a to a specific shuttle, application of $\mathsf{s2s}_1^*$ would be prioritized over the application (to the specific shuttle) of a rule $\mathsf{s2a}^*$ (Figure 7.6(c)) with $prio(\mathsf{s2a}^*) = 0$ and with only a trivial composed negative application condition true. Likewise, rule $\mathsf{s2s}_2^*$ in Figure 7.6(b) emulates the negative application condition $\neg\exists x_1$. Similar considerations are applicable to rules f2f and a2f. Of course, we will sill require a regular s2s rule (with $prio(\mathsf{s2s}) = 0$); a shuttle may also choose to stay in speed mode slow without a switch ahead.

In a similar fashion, we can replace the rules f2f and a2f by two rules $\mathsf{f2f}^*$ and $\mathsf{a2f}^*$ (with trivial application conditions true) and add rules $\mathsf{a2b}_1^*$ and $\mathsf{f2b}_1^*$, which are similar to $\mathsf{s2s}_1^*$ and have a priority of 1. Since the original rules f2f and a2f only have a single negative application condition each, we do not need another rule in either case. The remaining rules a2b, f2b, b2s remain unchanged; hence, $\mathcal{R} = \{\mathsf{s2s}, \mathsf{s2s}_1^*, \mathsf{s2s}_2^*, \mathsf{s2a}^*, \mathsf{f2f}^*, \mathsf{a2f}^*, \mathsf{a2b}, \mathsf{a2b}_1^*, \mathsf{f2b}, \mathsf{f2b}_1^*, \mathsf{b2s}\}$.

Note that this strategy may lead to unintended consequences if we analyze behavior of more than one shuttle. The notion of rule applicability with priorities used here is not specific to certain entities or matches. For example, applicability of $\mathsf{s2s}_1^*$ with $prio(\mathsf{s2s}_1^*) = 1$ will be prioritized over all potential rule applications of lower priority (such as s2a), even when those rule applications concern other shuttles in different situations. This is a concern in systems where concurrency needs to be taken into account.



(a) Graph rule $\mathsf{s2s}_1^*$ with $prio(\mathsf{s2s}_1^*) = 1$

(b) Graph rule $\mathsf{s2s}_2^*$ with $prio(\mathsf{s2s}_2^*) = 1$

(c) Graph rule $\mathsf{s2a}^*$ with $prio(\mathsf{s2a}^*) = 0$

Figure 7.6. – Updated graph rules with priorities

In this example, we want to verify a composed forbidden pattern $\mathcal{F} = \neg F_1$ (Figure 7.2(a), p. 172) that only forbids a shuttle on a switch in speed mode fast. We also use the same composed guaranteed pattern $\mathcal{H} = \neg H_1 \wedge \ldots \wedge \neg H_{15}$ as in Example 6.1 (p. 111) – or as in Example 7.1 (p. 170), but without the single fault assumption. △

This example has shown the potential of rule priorities to replace certain types of application conditions. Conversely, rule priorities may be replaced by suitable application conditions: in order to encode rules with priorities without using priorities, suitable (composed) negative application conditions for each rule of higher priority can be added to a rule of lower priority. In essence, the application conditions should prohibit the existence of any elements leading to the applicability of a rule of higher priority; then, the rule (of lower priority) can be applied. However, depending on the size of the rules and their common subgraphs – i.e. potential overlappings – those application conditions may grow to be rather large and unintuitive.

Considering graph rules with priorities requires an extension of our definition of k-inductive invariants based on the original Definition D.1 (p. 62). Basically, we replace the notion of rule application (previously denoted $\Rightarrow_{\mathcal{R}}$) with rule application via priorities (denoted $\Rightarrow_{\mathcal{R}}^{prio}$):

Definition 7.10 (k-inductive invariant and rule priorities)**.** *Given a typed graph transformation system with priorities $GTS = (TG, \mathcal{R}, prio)$ and graph constraints $\mathcal{F}$ and $\mathcal{H}$, $\mathcal{F}$ is a* k-inductive invariant *of GTS* under $\mathcal{H}$, *if, for all sequences of transformations to $\mathcal{R}$ $trans = G_0 \Rightarrow_{\mathcal{R}}^{prio} G_1 \Rightarrow_{\mathcal{R}}^{prio} \ldots \Rightarrow_{\mathcal{R}}^{prio} G_k$ it holds that:*

$$\forall z (0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H}) \Rightarrow (G_k \vDash \mathcal{F} \vee G_k \nvDash \mathcal{H})$$

In the restricted approach to k-inductive invariant checking, Lemma 6.10 (p. 123) rearranged Definition D.1 (p. 62) to a condition that is more directly verifiable. Likewise, we can rearrange Definition 7.10, again replacing the notion of rule applications:

Lemma 7.11 (k-inductive invariant and transformation sequences as counterexamples with priorities)**.** *Given a typed graph transformation system with priorities $GTS = (TG, \mathcal{R}, prio)$,* *a composed forbidden pattern $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$, and a composed guaranteed pattern $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$, $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$, if and only if the following holds for each k-sequence of transformations to $\mathcal{R}$ trans $= G_0 \Rightarrow_{\mathcal{R}}^{prio} \dots \Rightarrow_{\mathcal{R}}^{prio} G_k$:*

$$\exists u(G_k \vDash F_u) \Rightarrow (\exists z, v(0 \leq z \leq k \wedge G_z \vDash H_v) \vee \exists z, v(0 \leq z \leq k-1 \wedge G_z \vDash F_v))$$

Proof. This follows analogously to the proof of Lemma 6.10 (p. 123) based on Definition D.1 (p. 62). □

Taking priorities in $k-1$-bounded backward model checking into account requires updating the definition for bounded states spaces:

Definition 7.12 (bounded state space with priorities under constraint)**.** *Given a number $k \in \mathbb{N}$, we define the graph grammar's* state space with priorities bounded by k *(or* k-bounded state space*) as* REACH$_k(GG) = \{G \mid \exists n(0 \leq n \leq k \ \wedge \ G_0 \Rightarrow_{\mathcal{R}}^{n,prio} G)\}$. *Likewise, we define the* state space with priorities bounded by k under *a graph constraint $\mathcal{H}$ as* REACH$_k(GG, \mathcal{H}) = \{G \mid \exists n(0 \leq n \leq k \ \wedge \ G_0 \Rightarrow_{\mathcal{R}}^{n,prio} G)$ *such that all traversed graphs satisfy* $\mathcal{H}\}$.

This definition can then be rearranged, too:

Lemma 7.13 (k-bounded state spaces and transformation sequences as counterexamples)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o\in O} \neg S_o$ be graph constraints. For all graphs $G \in$ REACH$_k(GG, \mathcal{H})$ with graph grammars $GG \in$ IND$(GTS, \mathcal{S})$, we have $G \vDash \mathcal{F}$ if and only if $\mathcal{S} \vDash \mathcal{F}$ and the following holds for each sequence of transformations to $\mathcal{R}$ trans $= G_0 \Rightarrow_{b_1,m_1,m_1'}^{prio} \dots \Rightarrow_{b_n,m_n,m_n'}^{prio} G_n$ with $0 \leq n \leq k$:*

$$\exists u(G_n \vDash F_u) \Rightarrow (\exists z, v(0 \leq z \leq n \wedge G_z \vDash H_v) \vee \forall z(0 \leq z \leq n \Rightarrow \exists v(G_z \vDash S_v)))$$

Proof. This follows analogously to the proof of Lemma 6.11 (p. 124) based on Definition 2.28 (p. 36). □

In order to take rule priorities into account when verifying k-inductive invariants or performing $k-1$-bounded backward model checking, we leave the process of creating s/t-pattern sequences with the Seq-construction unchanged. Instead, we update the analysis of sequences, as described in the following section.

7.2.2. Analysis of Pattern Sequences

With respect to the verification of k-inductive invariants – or k-bounded backwards model checking – counterexamples are only valid if they follow the respective notion of rule applicability. Here, this includes priorities of graph rules. Priorities offer another opportunity to discard s/t-pattern sequences as counterexamples. In order to consider priorities in a symbolic fasion, we analyze (reductions of) source patterns in s/t-pattern sequences for implication of rule applicability constraints of rules of higher priority (than the rule connecting the source and subsequent target pattern). Successful implication means that a rule of higher priority would have been applicable in all satisfying transformation sequences. As such, the s/t-pattern sequence does not represent valid transformation sequences with respect to a notion of rule application including priorities.

Hence, we update Theorem T.2r by adding a third condition to discard s/t-pattern sequences:

Theorem T.2e-rp (k-inductive invariant checking with priorities)**.** *Let $GTS = (TG, (\mathcal{R}, prio))$ be a graph transformation system with priorities and $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$ and $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$ be a composed forbidden pattern and composed guaranteed pattern, respectively.*

$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ if, for all sequences $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ with $seq \in Seq_k(\mathcal{R}, \mathcal{F})$, one of the following conditions holds:

1. $\exists n,o(1 \leq n \leq k \wedge (src_{n|\varnothing} \models H_o \vee src_{z|\varnothing} \models F_v))$.
2. $\exists o(tar_{k|\varnothing} \models H_o)$.
3. $\exists n,b'(1 \leq n \leq k \ \wedge \ b' = \langle (L' \hookleftarrow K' \hookrightarrow R'), ac_{L'}, \text{true}\rangle \in \mathcal{R} \ \wedge \ prio(b') > prio(b_n) \ \wedge \ src_{n|\varnothing} \models \exists(i_{L'}, ac_{L'} \wedge \text{Appl}(b')))$.

Proof. According to Lemma 7.11 (p. 183), we need to show that for all k-sequences of transformations $G_0 \Rightarrow^{prio}_{\mathcal{R}} \dots \Rightarrow^{prio}_{\mathcal{R}} G_k$, it holds that:

$$\exists u(G_k \models F_u) \Rightarrow \exists z,v(0 \leq z \leq k \wedge G_z \models H_v) \vee \exists z,v(0 \leq z \leq k-1 \wedge G_z \models F_v)$$

Consider an arbitrary k-sequence of transformations to $\mathcal{R}$ (with corresponding graphs) $trans = G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ such that $\exists u(G_k \models F_u)$ with, for ease of reading, $F_u = F$. More specifically, $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_k,m_k,m'_k} G_k$ for rules $b_i \in \mathcal{R}$ and matches (comatches) m_i (m'_i) and $trans$ leads to F. By Theorem T.1r, there is a k-sequence of s/t-patterns $seq \in \text{Seq}_k(\mathcal{R}, F)$ with $trans \models seq$.

By precondition, one of the following is true:

1. There exist n,o with $1 \leq n \leq k$ such that $src_{n|\varnothing} \models H_o$ or $src_{n|\varnothing} \models F_o$. Because of $trans \models seq'$, we have $m_n \models src_n$ and, with $m_n : L_n \hookrightarrow G_{n-1}$ and Lemma 2.38, we gain $G_{n-1} \models src_{n|\varnothing}$ and, by implication of graph constraints (Definition 2.36), $G_{n-1} \models H_o$ or $G_{n-1} \models F_o$.
2. There exists o such that $tar_{k|\varnothing} \models H_o$. Because of $trans \models seq'$, we have $m'_k \models tar_k$ and, with $m'_k : R_k \hookrightarrow G_k$ and Lemma 2.38, we gain $G_k \models tar_{k|\varnothing}$ and, by implication of graph constraints (Definition 2.36), $G_k \models H_o$.
3. There exist n, b' with $1 \leq n \leq k$, $b' \in \mathcal{R}$ with $b' = \langle (L' \hookleftarrow K' \hookrightarrow R'), ac_{L'}, \text{true}\rangle$, and $prio(b') > prio(b_n)$ such that $src_{n|\varnothing} \models \exists(i_{L'}, ac_{L'} \wedge \text{Appl}(b'))$. Then, by Lemmas 2.38 (p. 43) and 2.30 (p. 37), there is a transformtion $G_{n-1} \Rightarrow_{b'} G'_n$. By Definition 7.6 (p. 181), this implies the absence of a transformation $G_{n-1} \Rightarrow^{prio}_{b_n,m_n,m'_n} G_n$ and, consequently, of a transformation sequence $G_0 \Rightarrow^{prio}_{b_1,m_1,m'_1} \dots \Rightarrow^{prio}_{b_k,m_k,m'_k} G_k$.

Thus, for all transformation sequences $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ with $\exists u(G_k \models F_u)$, there are n, o with $q \leq n \leq k$ such that $G_{n-1} \models H_o$ or $G_{n-1} \models F_o$, there is an o such that $G_k \models \mathcal{H}_o$, or $G_0 \Rightarrow^{prio}_{\mathcal{R}} \dots \Rightarrow^{prio}_{\mathcal{R}} G_k$ is not a transformation sequence (with respect to rule priorities). Then, by Lemma 7.11 (p. 183), $\mathcal{F}$ is a k-inductive invariant for GTS under H. □

Note that rule priorities are not encoded into s/t-pattern sequences – they are only considered during the analysis by the theorem's third condition. Hence, s/t-pattern sequences are still of the form $src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$, not $src_1 \Rightarrow^{prio}_{b_1} \dots \Rightarrow^{prio}_{b_k} tar_k$.

Example 7.14 (verification of 2-inductive invariants and rule priorities)**.** Consider the system described in Example 7.9 (p. 182). In order to apply Theorem T.2e-rp and verify $\neg F_1$ as a 2-inductive invariant, we need to compute $\text{Seq}^r_2(\mathcal{R}, F_1)$. An s/t-pattern sequence $seq_2 \in \text{Seq}^r_2(\mathcal{R}, F_1)$ is shown in Figure 7.7, with $seq_2 = \exists s_1 \Rightarrow_{\mathsf{s2a}^*} (\exists t_1, \exists s_2) \Rightarrow_{\mathsf{a2f}^*} \exists t_2$.

By the third condition of Theorem T.3e-rp, we need to consider whether $src_{1|\varnothing}$ or $src_{2|\varnothing}$ imply the rule applicability constraint of a rule of higher priority than $\mathsf{s2a}^*$ or $\mathsf{a2f}^*$, respectively, i.e. of $\mathsf{s2s}^*_1$, $\mathsf{s2s}^*_2$, $\mathsf{a2b}^*_1$, or $\mathsf{f2b}^*_1$. $\exists(i_{L_{\mathsf{s2s}^*_1}}, ac_{L_{\mathsf{s2s}^*_1}} \wedge \text{Appl}(\mathsf{s2s}^*_1))$ is depicted in Figure 7.8 – and indeed, $src_{1|\varnothing}$ implies $\exists(i_{L_{\mathsf{s2s}^*_1}}, ac_{L_{\mathsf{s2s}^*_1}} \wedge \text{Appl}(\mathsf{s2s}^*_1))$ and we can discard seq_2 as a potential counterexample. By the argument in the proof of Theorem T.2e-rp, any transformation sequence $trans = G_0 \Rightarrow_{\mathsf{s2a}^*,m_1,m'_1} G_1 \Rightarrow_{\mathsf{a2f}^*,m_2,m'_2} G_2$ satisfying seq_2 would also allow application of $\mathsf{s2s}^*_1$ to G_0. As a result, $trans' = G_0 \Rightarrow^{prio}_{\mathsf{s2a}^*,m_1,m'_1} G_1 \Rightarrow^{prio}_{\mathsf{a2f}^*,m_2,m'_2} G_2$ ($trans$ with priorities) is not a valid transformation sequence.

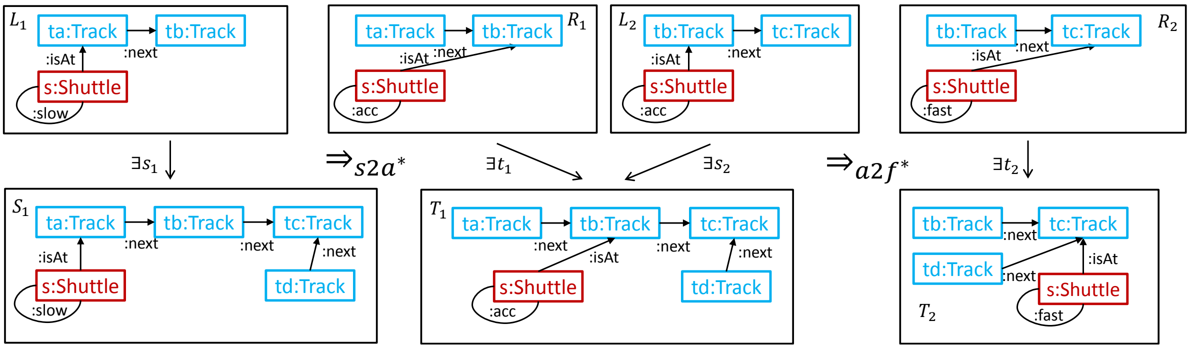


Figure 7.7. – S/t-pattern sequence $seq_2 = \exists s_1 \Rightarrow_{\mathsf{s2a^*}} (\exists t_1, \exists s_2) \Rightarrow_{\mathsf{a2f^*}} \exists t_2'$ with $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$

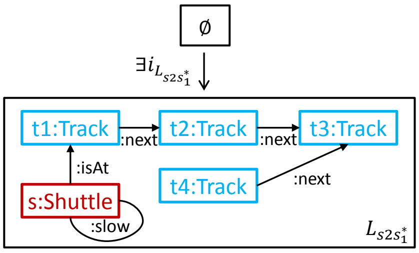


Figure 7.8. – Rule applicability constraint $\exists(i_{L_{\mathsf{s2s}_1^*}}, ac_{L_{\mathsf{s2s}_1^*}} \wedge \mathrm{Appl}(\mathsf{s2s}_1^*))$ of $\mathsf{s2s}_1^*$

Other s/t-pattern sequences in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$ will also be discarded by one of the condition in Theorem T.3e-rp: $\mathcal{F} = \neg F_1$ is indeed a 2-inductive invariant for $GTS = (TG, (\mathcal{R}, prio))$ under $\mathcal{H} = \neg H_1 \wedge \ldots \wedge \neg H_{15}$. △

Similarly, we can perform bounded backward model checking for graph transformation systems with priorities by adding a fourth condition to Theorem T.3r (p. 149).

Theorem T.3e-rp (k−1-bounded backward model checking with priorities)**.** *Let $GTS = (TG, (\mathcal{R}, prio))$ be a graph transformation system with priorities and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o \in O} \neg S_o$ be a composed forbidden pattern, composed guaranteed pattern, and composed start configuration pattern, respectively, with $\mathcal{S} \vDash \mathcal{F}$.*

For all graphs $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ and graph grammars $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$, we have $G \vDash \mathcal{F}$, if, for all sequences $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_n} tar_n$ with $seq \in SEQr_{k-1}(\mathcal{R}, \mathcal{F})$, one of the following conditions holds:

1. $\exists w, j(1 \le w \le n \wedge (src_{w|\varnothing} \vDash H_j))$.
2. $\exists j(tar_{k|\varnothing} \vDash H_j)$.
3. $\forall w(1 \le w \le n \Rightarrow \exists o(src_{n|\varnothing} \vDash S_o)) \wedge \exists o(tar_n \vDash S_o)$.
4. $\exists w, b'(1 \le w \le k \;\wedge\; b' = \langle (L' \hookleftarrow K' \hookrightarrow R), ac_{L'}, \mathrm{true}\rangle \in \mathcal{R} \;\wedge\; prio(b') > prio(b_w) \;\wedge\; src_{w|\varnothing} \vDash \exists(i_{L'}, ac_{L'} \wedge \mathrm{Appl}(b')))$.

Proof. According to Lemma 6.11 (p. 124), we need to show that for all sequences of transformations $G_0 \Rightarrow_{\mathcal{R}}^{prio} \ldots \Rightarrow_{\mathcal{R}}^{prio} G_n$ with $0 \le n \le k$, it holds that

$$\exists u(G_n \vDash F_u) \Rightarrow (\exists z, v(0 \le z \le n \wedge G_z \vDash H_v) \vee \forall z(0 \le z \le n \Rightarrow \exists v(G_z \vDash S_v)))$$

Consider an arbitrary n-sequence of transformations to $\mathcal{R}$ (with corresponding graphs) $trans = G_0 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n$ such that $0 \le n \le k$ and $\exists u(G_n \vDash F_u)$ with, for ease of reading,

$F_u = F$. More specifically, $\mathit{trans} = G_0 \Rightarrow_{b_1,m_1,m'_1} \dots \Rightarrow_{b_n,m_n,m'_n} G_n$ for rules $b_i \in \mathcal{R}$ and matches (comatches) m_i (m'_i) and *trans* leads to F.

By Theorem T.1r (p. 130), there is an n-sequence of s/t-patterns $\mathit{seq} \in \mathrm{Seq}^r_n(\mathcal{R}, F)$ with $\mathit{trans} \vDash \mathit{seq}$. Since $\mathit{seq} \in \mathrm{SEQ}^r_n(\mathcal{R}, \mathcal{F})$, by precondition, one of the following is true:

1. There exist w, j with $1 \le w \le n$ such that $\mathit{src}_{w|\varnothing} \vDash H_j$. Because of $\mathit{trans} \vDash \mathit{seq}'$, we have $m_w \vDash \mathit{src}_w$ and, with $m_w : L_w \hookrightarrow G_{w-1}$ and Lemma 2.38, we gain $G_{w-1} \vDash \mathit{src}_{w|\varnothing}$ and, by implication of graph constraints (Definition 2.36), $G_{w-1} \vDash H_j$.
2. There exists j such that $\mathit{tar}_{n|\varnothing} \vDash H_j$. Because of $\mathit{trans} \vDash \mathit{seq}'$, we have $m'_n \vDash \mathit{tar}_n$ and, with $m'_n : R_n \hookrightarrow G_n$ and Lemma 2.38, we gain $G_n \vDash \mathit{tar}_{n|\varnothing}$ and, by implication of graph constraints (Definition 2.36), $G_n \vDash H_j$.
3. For every reduced source pattern $\mathit{src}_{w|\varnothing}$ ($1 \le w \le n$) we have $\mathit{src}_{w|\varnothing} \vDash S_o$ ($\mathit{tar}_{n|\varnothing} \vDash S_o$) for some $o \in O$. Because of $\mathit{trans} \vDash \mathit{seq}'$, we have $m_w \vDash \mathit{src}_w$ and, with $m_w : L_w \hookrightarrow G_{w-1}$ and Lemma 2.38, we gain $G_{w-1} \vDash \mathit{src}_{w|\varnothing}$ and, by implication of graph constraints (Definition 2.36), $G_{w-1} \vDash S_o$ for the respective w and o. The same reasoning applies to $\mathit{tar}_{n|\varnothing}$, implying $G_n \vDash S_o$ for an $o \in O$.
4. There exist w, b' with $1 \le w \le n$, $b' \in \mathcal{R}$ with $b' = \langle (L' \hookleftarrow K' \hookrightarrow R), \mathit{ac}_{L'}, \mathrm{true}\rangle$, and $\mathit{prio}(b') > \mathit{prio}(b_w)$ such that $\mathit{src}_{w|\varnothing} \vDash \exists(i_{L'}, \mathit{ac}_{L'} \wedge \mathrm{Appl}(b'))$. Then, by Lemmas 2.38 (p. 43) and 2.30 (p. 37), there is a transformtion $G_{w-1} \Rightarrow_{b'} G'_w$. By Definition 7.6 (p. 181), this implies the absence of a transformation $G_{w-1} \Rightarrow^{\mathit{prio}}_{b_w,m_w,m'_w} G_n$ and, consequently, of a transformation sequence $G_0 \Rightarrow^{\mathit{prio}}_{b_1,m_1,m'_1} \dots \Rightarrow^{\mathit{prio}}_{b_n,m_n,m'_n} G_n$.

Hence, all transformation sequences of length between 0 and k leading to F have violations of $\mathcal{H}$, do not contain graphs satisfying $\mathcal{S}$, or are invalid transformation sequences with respect to priorities. Then, by Lema 7.13 (p. 184), $\mathcal{F}$ is satisfied by all graphs in the $k-1$-bounded state spaces of graph grammars with priorities induced by GTS and $\mathcal{S}$. □

Theorems T.4r (p. 153) and T.5r (p. 155) can be modified in a similar fashion.

7.2.3. Implementation

Taking priorities and hence, the additional conditions of Theorems T.2e-rp (p. 184) and T.3e-rp (p. 186), into account requires extending the analysis of source patterns. In the functions createSequences (Algorithm 6.8 (p. 159)) and extendSequences (Algorithm 6.9 (p. 160)), the call to discardPattern($\mathit{src}_{|\varnothing}$, $\mathcal{C}$) is changed to discardPattern($\mathit{src}_{|\varnothing}$, $\mathcal{C} \cup$ applConstraints($(\mathcal{R}, \mathit{prio}), b$)). The function applConstraints is described in Algorithm 7.4. The updated functions are shown in Algorithms 7.5 and Algorithms 7.6, with the changes in lines 10 and 13, respectively.

Algorithm 7.4: applConstraints($(\mathcal{R}, \mathit{prio}), b$)

```
desc.  : creates rule applicability constraints for all rules of higher priority
input  : a set $\mathcal{R}$ of graph rules, their priority function $prio$, and a graph rule $b$
output: a set of rule applicability constraints of rules in $\mathcal{R}$ with a higher priority
        than $b$

results ← ∅
foreach $b' \in \mathcal{R}$ with $b' = \langle (L' \hookleftarrow K' \hookrightarrow R'), ac_{L'}, \mathrm{true}\rangle$ do
    if $prio(b') > prio(b)$ then
        results ← results ∪ {$\exists(i_{L'}, \mathrm{Appl}(b') \wedge ac_{L'})$}
return results
```

```
Algorithm 7.5: createSequences($\mathcal{R}$, F, $\mathcal{C}_1$, $\mathcal{C}_2$)
  desc.   : implements $\mathrm{Seq}_1^r(\mathcal{R}, F)$ with optimizations and rule priorities
  input   : a set $\mathcal{R}$ of graph rules with priorities $prio$, a forbidden pattern
            $F = \exists(i_P, ac_P)$, sets $\mathcal{C}_1$ and $\mathcal{C}_2$ of patterns
  output: $\mathrm{Seq}_1^r(\mathcal{R}, F)$ minus sequences discarded for violations of $\mathcal{C}_1$ or $\mathcal{C}_2$
  $results \leftarrow \varnothing$
  foreach $b \in \mathcal{R}$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ do
      foreach $tar$ in $\bigvee_{j \in J} tar_j = \mathrm{Shift}(i_R, \exists i_P)$ with $tar = \exists t$ do     /* SC$_1$-1 */
          if not discardPattern($tar_{|\varnothing}$, $\mathcal{C}_1$) then                                      /* analysis */
              $tar \leftarrow \exists(t, \mathrm{Shift}(p, ac_P))$                                              /* SC$_1$-1 */
              if not discardPattern($tar_{|\varnothing}$, $\mathcal{C}_1$) then                                  /* analysis */
                  $src' \leftarrow \mathrm{L}(b, tar)$                                                          /* SC$_1$-2 */
                  if $src'$ has the form $\exists(s, ac')$ then                                                 /* SC$_1$-3 */
                      $src \leftarrow \exists(s, ac' \wedge \mathrm{Shift}(s, ac_L \wedge \mathrm{Appl}(b)))$      /* SC$_1$-3 */
                      if not discardPattern($src_{|\varnothing}$,
                      $\mathcal{C}_2 \cup$ applConstraints(($\mathcal{R}$, $prio$), $b$)) then                   /* analysis */
                          $results \leftarrow results \cup \{src \Rightarrow_b tar\}$                           /* SC$_1$-4 */
  return $results$                                                                                              /* SC$_1$-5 */
```

```
Algorithm 7.6: extendSequences($\mathcal{R}$, $sequences$, $\mathcal{C}$)
  desc.   : implements $\mathrm{Seq}_{k+1}^r(\mathcal{R}, \neg\mathcal{F})$ (based on the result of $\mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$) with
            optimizations and rule priorities
  input   : $\mathrm{Seq}_k^r(\mathcal{R}, F)$, i.e. a set of sequences $sequences$ of the form
            $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$ with $src_1 = \exists(s_1, ac_{S_1})$, a set $\mathcal{R}$ of graph rules
            with priorities $prio$, a set $\mathcal{C}$ of patterns
  output: $\mathrm{Seq}_{k+1}^r(\mathcal{R}, F)$ minus sequences discarded for violations of $\mathcal{C}$
  $results \leftarrow \varnothing$
  foreach $seq \in sequences$ with $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k$ do
      foreach $b \in \mathcal{R}$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ do
          foreach $tar$ in $\bigvee_{j \in J} tar_j = \mathrm{Shift}(i_R, \exists i_{S_1})$ with $tar = \exists t$ do     /* SC$_\mathrm{k}$-1 */
              if not discardPattern($tar_{|\varnothing}$, $\mathcal{C}$) then                                        /* analysis */
                  $ac_T \leftarrow \mathrm{Shift}(s', ac_{S_1})$                                                      /* SC$_\mathrm{k}$-1 */
                  $tar \leftarrow \exists(t, ac_T)$                                                                    /* SC$_\mathrm{k}$-1 */
                  if not discardPattern($tar_{|\varnothing}$, $\mathcal{C}$) then                                    /* analysis */
                      $src_1^+ \leftarrow \exists(s' \circ s_1, ac_T)$                                                  /* SC$_\mathrm{k}$-1$^+$ */
                      $src' \leftarrow \mathrm{L}(b, tar)$                                                              /* SC$_\mathrm{k}$-2 */
                      if $src'$ has the form $\exists(s, ac')$ then                                                     /* SC$_\mathrm{k}$-3 */
                          $src \leftarrow \exists(s, ac' \wedge \mathrm{Shift}(s, ac_L \wedge \mathrm{Appl}(b)))$         /* SC$_\mathrm{k}$-3 */
                          if not discardPattern($src_{|\varnothing}$,
                          $\mathcal{C}_2 \cup$ applConstraints(($\mathcal{R}$, $prio$), $b$)) then                      /* analysis */
                              $results \leftarrow results \cup$
                              $\{src \Rightarrow_b (tar, src_1^+) \Rightarrow_{b_1} \ldots \Rightarrow tar_k\}$             /* SC$_\mathrm{k}$-4 */
  return $results$                                                                                                  /* SC$_\mathrm{k}$-5 */
```

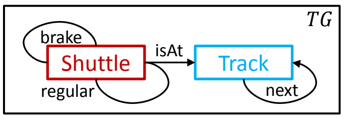


Figure 7.9. – Type graph

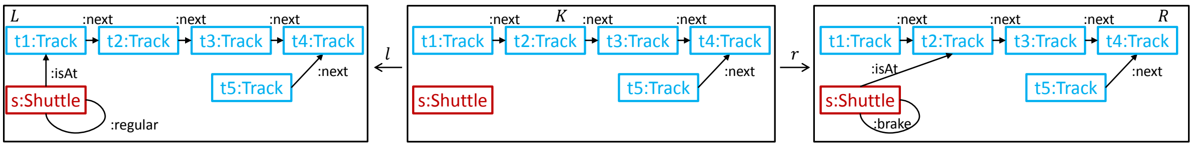


Figure 7.10. – Graph rule brake

7.3. Partial Negative Application Conditions

One of the most costly factors of the approach and algorithms is the transformation of application conditions over morphisms via the Shift-construction. Given an application condition $ac = \exists(a : A \hookrightarrow P)$ and a morphism $b : A \hookrightarrow B$, computing $\text{Shift}(b, ac)$ amounts to finding all occurrences of all subgraphs of B in P (or of P in B). Since the number of subgraphs of a graph grows exponentially with the number of its nodes and edges, it is desirable to avoid or optimize this computation where possible. To achieve this, this section presents *partial negative application conditions*, which contribute to **Appl.-performance**.

The Shift-construction is used in the following steps of the restricted approach:

- in step SC$_1$-1 of the Seq-construction when computing $\text{Shift}(i_R, C)$,
- in step SC$_k$-1 of the Seq-construction when computing $\text{Shift}(s'_j, ac_{S_1})$ and – implicitly – when finding injective and jointly surjective morphism pairs (t_j, s'_j),
- in steps SC$_1$-3 and SC$_k$-3 of the Seq-construction when computing $\text{Shift}(s_j, ac_L \wedge \text{Appl}(b))$, and
- when determining implication of graph patterns by application of Theorem 6.8 (p. 120) and, as a result, in the application of Theorems T.2r (p. 143) and T.3r (p. 149).

Example 7.15 (running example)**.** In order to demonstrate the problem of Shift with respect to computational effort, we consider a fragment of a slightly different example. Figure 7.9 shows a type graph where shuttles can drive in one of two modes: braking and regular, both denoted by an edge of type brake or regular, respectively.

Figure 7.10 then shows a graph rule brake that requires shuttles to brake when approaching a switch three tracks ahead. The property we want to verify for this small example system is depicted in Figure 7.11: a shuttle should not brake ($\neg\exists(i_{P_1^F}, \dots)$) unless it is positioned on a switch ($\neg\exists x_1$) or a switch is directly ahead ($\neg\exists x_2$). While unnecessary braking is not usually as much of a safety concern as a dangerously high velocity, it is inefficient – hence the verification. An extended form of this example is shown in Section C.1.5 of Appendix C and will be a focus of the evaluation later. △

Example 7.16 (composed negative application conditions and Shift-construction)**.** If we want to verify $\mathcal{F} = \neg F_1$ (Figure 7.11) as a 1-inductive invariant of $GTS = (TG, \mathcal{R})$ with $\mathcal{R} = \{\text{brake}\}$, we will have to compute $\text{Seq}_1^r(\mathcal{R}, F_1)$. In step SC$_1$-1 of the Seq-construction, this requires computing $\text{Shift}(i_R, F_1)$. Intuitively, this means finding overlappings of P_1^F and the rule's right rule side R – and then, to find overlappings between X_1 and the remainder of R and X_2 and the remainder of R, respectively.

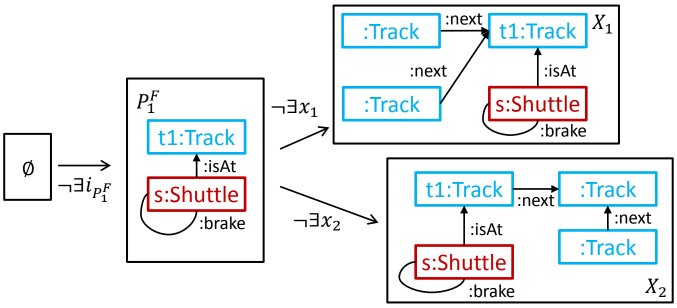


Figure 7.11. – Composed forbidden pattern $\mathcal{F} = \neg F_1 = \neg\exists(i_{P_1^F}, \neg\exists x_1 \wedge \neg\exists x_2)$

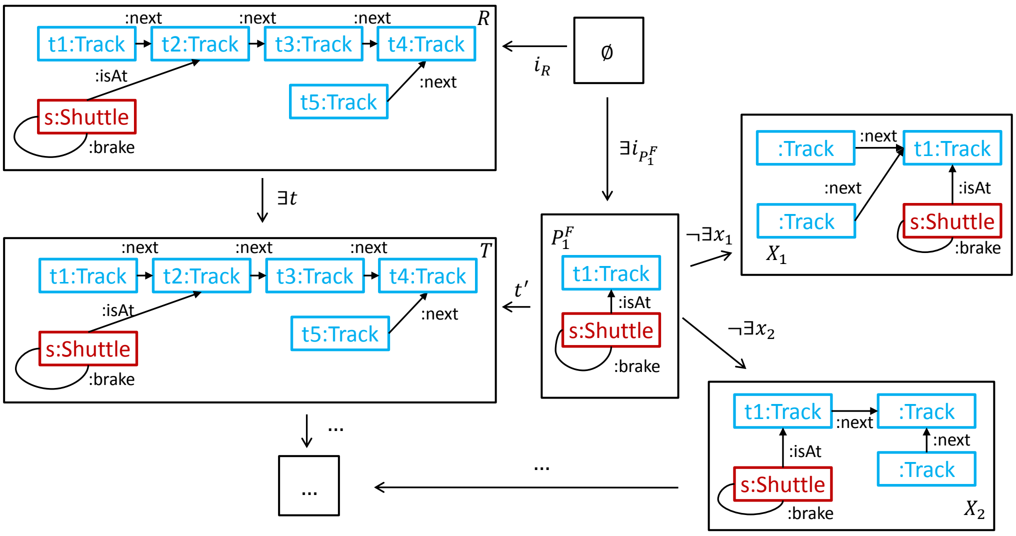


Figure 7.12. – Fragments of an example target pattern $tar = \exists(t, \mathrm{Shift}(t', \neg\exists x_1 \wedge \neg\exists x_2))$

Figure 7.12 shows fragments of one possible target pattern resulting from the comutation of $\text{Shift}(i_R, \exists(i_{P_1^F}, \dots))$: T describes one possible overlapping of P_1^F and R. It corresponds to a target pattern $\exists(t, \text{Shift}(t', \neg\exists x_1 \wedge \neg\exists x_2))$, where the forbidden patterns composed negative application condition still has to be shifted to the context of the graph T. Formally, shifting $\neg\exists x_2$ requires finding all graphs T_i' and injective morphism pairs $(x_{2i}' : X_2 \hookrightarrow T_i', t_{2i}' : T \hookrightarrow T_i')$ such that $t_{2i}' \circ t' = x_{2i}' \circ x_2$. Intuitively, we look for overlappings (possibly including the empty overlapping) between X_2 and R while preserving the overlapping of P_1^F and R in T. The same holds for $\neg\exists x_1$ and X_1, respectively. In this example, shifting $\neg\exists x_2$ to the context of T means to look for possibilities to extend T such that the braking shuttle s appears in the situation described by X_2: one track ahead of a switch.

This means that we have to consider all possibilities to map all subgraphs of $X_2 \setminus x_2(P_1^F)$ (injectively) to subgraphs of $T \setminus t'(P_1^F)$. For reasons of simplicity, we will leave edges aside here for purposes of demonstrating the problem's complexity. Even without considering edges, we get

$$\binom{4}{0} \cdot 0! + \binom{4}{1} \cdot 1! + \binom{4}{1} \cdot 1! + \binom{4}{2} \cdot 2! = \frac{4!}{4!} + 2 \cdot \frac{4!}{3!} + \frac{4!}{2!} = 21$$

graphs and corresponding injective morphism pairs already, with the same number for shifting $\neg x_1$.

Now, let us consider a modified brake rule that looks for switches four tracks ahead instead of just three – and, to match the rule, a forbidden pattern that only allows a shuttle to brake if there is a switch one or two tracks ahead. Given $\exists x_2$ as a negative application condition with three additional tracks in X_2, there is a total of

$$\binom{5}{0} \cdot 0! + \binom{5}{1} \cdot 1! + \binom{5}{1} \cdot 1! + \binom{5}{1} \cdot 1! + \binom{5}{2} \cdot 2! + \binom{5}{2} \cdot 2! + \binom{5}{2} \cdot 2! + \binom{5}{3} \cdot 3! = \frac{5!}{5!} + 3 \cdot \frac{5!}{4!} + 3 \cdot \frac{5!}{3!} + \frac{5!}{2!} = 136$$

graphs and inejctive morphism pairs – and again, we have not even considered edges yet. △

In general, given two graphs consistsing of n and m nodes (with $n > m$) of the same type, respectively, the number of overlappings is

$$ol(n, m) = \sum_{i=0}^{m} \binom{m}{i} \frac{n!}{(n-i)!} = \sum_{i=0}^{m} \binom{m}{i} \binom{n}{i} i!.$$

As a result, the number of negative application conditions in composed negative applications of target patterns can grow rather large for cases with unfortunate combinations of right rule sides and composed negative application conditions of forbidden patterns. While some of the fragments may be meaningless for the purpose of verification, the algorithms shown in the restricted approach cannot filter conditions out before computing them – and even if such filtering were possible, it may not be more efficient.

Of course, this problem is not limited to composed negative application conditions in target patterns: given forbidden patterns with large positive contexts that are similar – in terms of node and edge types – to right rule sides, similar problems arise for the positive context of target patterns. However, the extension described in this section focuses on addressing the problem of large composed negative application conditions rather than target patterns, for reasons explained below.

In this example above and generally in steps SC_1-1 and SC_k-1 of the Seq-construction, Shift is used twice: first, to create the target patterns' existential conditions – i.e. their positive context – then, to transform the composed negative application conditions to the target patterns' contexts. Both steps play a different role with respect to the interpretation of the verification results. The first execution of Shift is required to determine the interaction of rules and forbidden patterns and creates a symbolic encoding of which elements are present in potential counterexamples. The second Shift further restricts these symbolic encodings by composed negative application conditions, which determine the elements and structures required to be absent. Often, counterexamples can be discarded because of the information contained in the positive contexts of source or target patterns already. Likewise, positive context is usually more helpful in order to understand why a forbidden pattern is not a k-inductive invariant.

For example, let us consider a reduced target pattern $tar_{|\varnothing} = \exists(i_T : \varnothing \hookrightarrow T, ac_T)$ and, for purposes of demonstrating the analysis of pattern implication, a guaranteed pattern $H = \exists(i_P, ac_P)$ where both ac_T and ac_P are composed negative application conditions. Then, by implication of patterns (Theorem 6.8 (p. 120)), if $ac_P = \text{true}$, we have $tar_{|\varnothing} \vDash H$ if there is an injective morphism $p : H \hookrightarrow T$. We do not need to consider ac_T, no matter how complex the condition is. If ac_T is not trivially true, we have to consider both the positive contexts and the composed negative application conditions ac_T and ac_P.

This leads to the following goals when dealing with the Shift-construction and composed negative application conditions:

1. When creating the target patterns in steps $\mathrm{SC_1}$-1 and $\mathrm{SC_k}$-1 with the Shift-construction, defer shifting composed negative application conditions for each target pattern $tar = \exists(t_j, \dots)$ until its existential part $\exists t_j$ has been compared to guaranteed and forbidden patterns with trivial composed negative application conditions (i.e. true).
2. Only shift composed negative application conditions (by the Shift-construction) to the extent necessary for verification.

To defer computation of shifted composed negative application conditions (1), we require a formalization of conditions not yet shifted to their new context. To shift the conditions at a later point (2), we require a construction for this formalization.

In order to get an intuitive view for the solution, consider a typical Shift-construction (in the restricted approach) depicted below – a pattern $\exists(i_P, \neg\exists x)$ to be shifted over the morphism i_R.

$$\begin{tikzcd}
\varnothing \arrow[r, hook, "i_R"] \arrow[d, hook, "i_P"] & R & & \varnothing \arrow[r, hook, "i_R"] \arrow[d, hook, "i_P"] & R \arrow[d, hook, "t_j"] \\
P \arrow[d, hook, "x"] & & & P \arrow[r, hook, "t'_j"] \arrow[d, hook, "x"] & T_j \\
X & & & X &
\end{tikzcd}$$

The left diagram shows the premise, the right diagram presents the construction's result: $\mathrm{Shift}(i_R, \exists(i_P, \neg\exists x)) = \bigvee_{j \in J} \exists(t_j, \mathrm{Shift}(t'_j, \neg\exists x))$ for injective and jointly surjective morphism pairs (t_j, t'_j). Now, to defer computing the second shift, we need a representation of $\mathrm{Shift}(t'_j, \neg\exists x)$ as an application over T_j that, unless it is fully transformed via Shift, only refers to a subgraph of T_j – in particular, to the image of P under t'_j. In order to express application conditions that only refer to a part of the context they are defined over, we use *partial application conditions*, which are based on *injective partial morphisms.*

Definition 7.17 (injective partial morphism *[1]*, originally partial monomorphism [Pen09])**.** *An* injective partial morphism *$p : A \hookrightarrow B$ is a 2-tuple $p = \langle a, b \rangle$ of injective morphisms a, b with $dom(a) = dom(b)$, $dom(p) = codom(a)$, and $codom(p) = codom(b)$. The* interface *of p refers to the common domain of a and b, i.e., $iface(p) = dom(a) = dom(b)$. An injective partial morphism $p = \langle a, b \rangle$ is said to be an injective total morphism b, if a is bijective.*

$$\begin{tikzcd}
A' \arrow[r, hook, "a"'] \arrow[d, hook, "b"'] & A \arrow[dl, hook, harpoon, "p=\langle a,b\rangle"] \\
B &
\end{tikzcd}$$

An injective partial morphism $p : A \lhook\joinrel\rightharpoonup B$ describes a mapping of the subgraph A' of A to B. Note that any injective total morphism $b : A \hookrightarrow B$ can be written as an injective partial morphism $p = \langle id_A, b\rangle$. Regarding the situation described above, we can use injective partial morphisms $p = \left\langle t'_j, x\right\rangle$ to describe an application condition over T_j that only has P as its context. In particular, $\exists p$ would be such a partial application condition. More generally, they are defined as follows:

Definition 7.18 (partial application conditions *[1]*, based on application conditions [EGH⁺14])**.** *A* partial application condition *is inductively defined as follows:*

1. *For every graph A, true is a partial application condition over A.*
2. *For every injective partial morphism $p : A \lhook\joinrel\rightharpoonup B$ with $p = \langle a, b\rangle$ and injective morphisms $a : A' \hookrightarrow A$ and $b : A' \hookrightarrow B$ and every partial application condition ac over B, $\exists(p, ac)$ is an application condition over A.*
3. *For partial application conditions ac, ac_i over A with $i \in I$ (for all index sets I), $\neg ac$ and $\bigwedge_{i\in I} ac_i$ are partial application conditions over A.*

$$\begin{tikzcd}
A' \arrow[r, hook, "a"'] \arrow[d, hook, "b"'] & A \arrow[dl, hook, harpoon, "p=\langle a,b\rangle"] \\
ac \triangleright B &
\end{tikzcd}
\qquad
\begin{tikzcd}
A' \arrow[r, hook, "a"'] \arrow[d, hook, "b"'] & A \arrow[dl, hook, harpoon, "p=\langle a,b\rangle"] \arrow[dr, "g"] & \\
ac \triangleright B \arrow[rr, hook, "q\,\models\,ac"'] & & G
\end{tikzcd}$$

Satisfiability *of partial application conditions is inductively defined as follows:*

1. *Every morphism satisfies true.*
2. *A morphism $g : A \to G$ satisfies $\exists(p, ac)$ over A with $p : A \lhook\joinrel\rightharpoonup B$ and $p = \langle a, b\rangle$ if there exists an injective $q : B \hookrightarrow G$ such that $q \circ b = g \circ a$ and q satisfies ac.*
3. *A morphism $g : A \to G$ satisfies $\neg ac$ over A if g does not satisfy ac and g satisfies $\bigwedge_{i\in I} ac_i$ over A if g satisfies each ac_i $(i \in I)$.*

We write $g \models ac$ to denote that the morphism g satisfies ac.

Two partial application conditions ac and ac' are equivalent*, denoted by $ac \equiv ac'$, if for all morphisms $g : A \to G$, $g \models ac$ if and only if $g \models ac'$.*

If all morphisms involved in a partial application condition are total morphisms, we say that it is a total application condition, or simply application condition.

$\exists p$ abbreviates $\exists(p, \text{true})$. $\forall(p, ac)$ abbreviates $\neg\exists(p, \neg ac)$.

In order to use partial application conditions in the restricted approach, we have to extend several concepts of its restricted formal model and symbolic encoding. First, we extend composed negative application conditions to composed partial negative application conditions.

Definition 7.19 ((composed) partial negative application condition *[1]*)**.** *A partial negative application condition is a partial application condition of the form $ac = \neg\exists p$ for an injective partial morphism p. A composed partial negative application condition is a partial application condition of the form $ac = \bigwedge_{i\in I} \neg p_i$ for injective partial morphisms p_i. A (composed) partial negative application condition with only total graph morphisms is a (composed) negative application condition or (composed) total negative application condition.*

With these definitions, partial application conditions and composed partial negative application conditions are more general notions of application conditions and composed negative application conditions, respectively. Conversely, any composed (total) negative application conditions is a composed partial negative application condition and any (total) application condition is a partial application condition. Furthermore, a composed partial negative application condition may consist of partial and total negative application conditions. This also holds for partial application conditions in general: since partial application conditions may contain total morphisms, a partial application condition may contain total application conditions in its nesting or operands.

Definitions 7.18 and 7.19 allow us to define partial application conditions, but do not specify the relation between partial application conditions and (total) application conditions or how partial application conditions can be used to equivalently describe application conditions to be shifted over a morphism. However, the connection is implicitly encoded in the definition of satisfiability of partial application conditions. Consider the respective diagram: a morphism $g : A \rightarrow G$ satisfies $\exists(p, ac)$ if there is an injective morphism $q : B \hookrightarrow G$ such that $q \circ b = g \circ a$. This is equivalent to $g \circ a$ satisfying $\exists(b, ac)$ – and, by the Shift-lemma, to g satisfying $\mathrm{Shift}(a, ac)$. This connection is formalized in the following lemma and construction.

Lemma 7.20 (PShift-lemma *[1]*)**.** *There is a construction* PShift *such that, for every injective morphism* $p' : P' \hookrightarrow P$ *and every partial application condition* ac *over* P', $\mathrm{PShift}(p', ac)$ *transforms* ac *via* p' *into a partial application condition over* P *such that, for each injective morphism* $n : P \hookrightarrow H$*, it holds that* $n \circ p' \vDash ac \Leftrightarrow n \vDash \mathrm{PShift}(p', ac)$ *and* $\mathrm{PShift}(p', ac) \equiv \mathrm{Shift}(p', ac)$*.*

$$\begin{tikzcd}
ac \rhd P' \arrow[rr, hook, "p'"] \arrow[dr, hook, "n \circ b"'] & \arrow[d, phantom, "="] & P \lhd \mathrm{PShift}(p',ac) \arrow[dl, hook', "n"] \\
& H &
\end{tikzcd}
\qquad
\begin{tikzcd}
ac \rhd P' \arrow[rr, hook, "p'"] \arrow[dr, hook, "n \circ p'"'] & \arrow[d, phantom, "="] & P \lhd \mathrm{Shift}(p',ac) \arrow[dl, hook', "n"] \\
& H &
\end{tikzcd}$$

Construction (PShift-construction *[1]*)**.** *The* PShift*-construction is defined as follows:*

$$\begin{tikzcd}
P' \arrow[r, hook, "p'"] \arrow[d, hook, "a"] & P \arrow[dl, "c=\langle p',a \rangle"] \\
ac \rhd C &
\end{tikzcd}$$

1. $\mathrm{PShift}(p', \mathrm{true}) = \mathrm{true}$.
2. $\mathrm{PShift}(p', \exists(a, ac)) = \exists(c, ac)$ *with* $c = \langle p', a \rangle$*.*
3. $\mathrm{PShift}(p', \neg ac) = \neg\,\mathrm{PShift}(p', ac)$*.*
4. $\mathrm{PShift}(p', \bigwedge_{i \in I} ac_i) = \bigwedge_{i \in I} \mathrm{PShift}(p', ac_i)$*.*

Proof. (Taken (with minor modifications) from the proof of Lemma 2.32 [EGH⁺14].)

We will prove $n \circ p' \vDash ac \Leftrightarrow n \vDash \mathrm{PShift}(p', ac)$ by structural induction:

Base case. For ac = true, the statement holds.

Inductive step. For an application condition $\mathrm{ac} = \exists(a, \mathrm{ac}')$, we have to show $n \circ p' \vDash \mathrm{ac} \Leftrightarrow n \vDash \mathrm{PShift}(p', \exists(a, \mathrm{ac}'))$.

Only if. Let $n \circ p' \vDash \exists(a, \mathrm{ac}')$. Consequently, there exists an injective morphism $q : C \hookrightarrow H$ such that $n \circ p' = q \circ a$ and $q \vDash \mathrm{ac}$. Thus, by satisfiability of partial application conditions, we have $n \vDash \exists(c, \mathrm{ac})$ with $c = \langle p', a \rangle$ and by construction we get $n \vDash \mathrm{PShift}(p', \exists(a, \mathrm{ac}'))$.

If. Let $n \vDash \mathrm{PShift}(p', \exists(a, \mathrm{ac}'))$. By construction, we have $n \vDash \exists(c, \mathrm{ac})$ with $c = \langle p', a \rangle$. By satisfiability of partial application conditions, we conclude the existence of an injective morphisms $q : C \hookrightarrow H$ such that $n \circ p' = q \circ a$ and $q \vDash \mathrm{ac}$. Thus, we have $p' \vDash \mathrm{ac}$, which concludes the inductive proof.

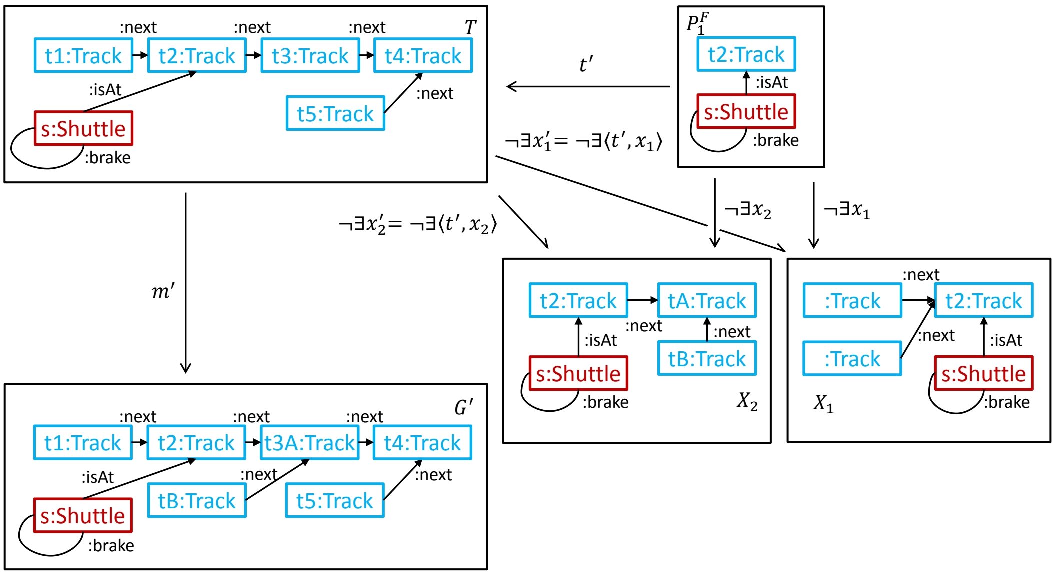


Figure 7.13. – Composed partial negative application condition $\neg\exists x_1' \wedge \neg\exists x_2'$ with $x_1' = \langle t', x_1\rangle$ and $x_2' = \langle t', x_2\rangle$

By the Shift-lemma, we have $n \circ p' \vDash ac \Leftrightarrow n \vDash \text{Shift}(p', ac)$, implying $n \vDash \text{Shift}(p', ac) \Leftrightarrow n \vDash \text{PShift}(p', ac)$ and, consequently, $\text{PShift}(p', ac) \equiv \text{Shift}(p', ac)$. □

The equivalence $n \circ p' \vDash ac \Leftrightarrow n \vDash \text{PShift}(p', ac) \Leftrightarrow n \vDash \text{Shift}(p', ac)$ has two important results. First, we can use partial application conditions as a symbolic encoding for application conditions to be shifted over a morphism without actually executing Shift. Second, we can still execute the construction at a later point, if necessary.

Example 7.21 (partial application condition)**.** Consider Example 7.16 (p. 189): we described a target pattern $\exists(t, \text{Shift}(t', \neg\exists x_1 \wedge \neg\exists x_2))$ (Figure 7.12 (p. 190)), where execution of the Shift-construction would lead to more than 42 conjunctively joined negative application conditions in the resulting composed negative application condition. Instead, we apply the PShift-construction, shown in Figure 7.13:

$$\text{PShift}(t', \neg\exists x_1 \wedge \neg\exists x_2) = \neg\exists x_1' \wedge \neg\exists x_2' \text{ with } x_1' = \langle t', x_1\rangle \text{ and } x_2' = \langle t', x_2\rangle$$

where $\neg\exists x_1' \wedge \neg\exists x_2'$ is a composed partial negative application condition. In addition, by Lemma 7.20 (p. 194), $\neg\exists x_1' \wedge \neg\exists x_2'$ is equivalent to $\text{Shift}(t', \neg\exists x_1 \wedge \neg\exists x_2)$, which also implies equivalence of $\exists(t, \text{Shift}(t', \neg\exists x_1 \wedge \neg\exists x_2))$ and $\exists(t, \neg\exists x_1' \wedge \neg\exists x_2')$. Essentially, we have replaced $\text{Shift}(\neg\exists x_1 \wedge \neg\exists x_2)$ by an equivalent composed partial negative application condition. If necessary, we can still compute the Shift-construction to get its explicit result. Admittedly, partial application conditions are not ideal for manual inspection – their primary purpose here is to reduce computational effort and the size of conditions.

Figure 7.13 also shows a morphism $m' : T \hookrightarrow G$. By definition of satisfiability, we have $m' \nvDash \neg\exists x_2'$: we can find an injective morphism $q : X_2 \hookrightarrow G$ such that $q \circ x_2 = m' \circ t'$ (by mapping s to s, t2 to t2, tA to t3A, tB to tB, and mapping the edges accordingly). (If tB in G' were attached to t5 or t1, for example, we would get $m' \vDash \neg\exists x_1' \wedge \neg\exists x_2'$.)

Then, by Lemma 7.20 (p. 194), we know that m' does not satisfy $\text{Shift}(t', \neg\exists x_1 \wedge \neg\exists x_2)$. In particular, $\neg\exists m'$ would be one of the negative application conditions in the composed negative application condition $\text{Shift}(t', \neg\exists x_2)$. △

Of course, using composed partial negative application conditions and the PShift-construction in this fashion is only useful if we can still perform the steps of the restricted approach – at least to some extent. This concerns other steps of the Seq-construction and the analysis of reduced source and target patterns for implication of forbidden or guaranteed patterns.

In Example 7.21, we have created a target pattern with a nested composed partial negative application condition. Since target patterns were originally required to have (total) composed negative application conditions (Definition 6.13 (p. 127)), we need to extend their definition. The same applies to the other elements of our symbolic encoding: source pattern and target/-source patterns.

7.3.1. Symbolic Encoding

In source and target patterns, composed negative application conditions are now allowed to be composed partial negative application conditions. As a result, source and target patterns themselves are partial application conditions instead of (total) application conditions.

Definition 7.22 (source pattern with composed partial negative application condition [3])**.** *Given a graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$, a* source pattern *over b is a partial application condition over the left side L of the form $src = \exists(s, ac_S)$ with s being a total injective morphism and ac_S a composed partial negative application condition.*

Definition 7.23 (target pattern with composed partial negative application condition [3])**.** *Given a graph rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$, a* target pattern *over b is a partial application condition over the right side R of the form $tar = \exists(t, ac_T)$ with t being a total injective morphism and ac_T a composed partial negative application condition.*

Note that these definitions follow the idea of combining right rule sides and forbidden patterns in step SC$_1$-1 (or leftmost source patterns in step SC$_\text{k}$-1) with respect to their existential conditions, but deferring the transformation of composed negative application conditions. Thus, morphisms s in source patterns $\exists(s, ac_S)$ are still required to be total morphisms and, consequently, $\exists s$ would be a total application condition. The same holds for morphisms t in target patterns $\exists(t, ac_T)$.

We will further modify the definition of target/source patterns accordingly:

Definition 7.24 (target/source pattern with composed partial negative application condition)**.** *Given rules $b_1 = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), ac_{L_1}, ac_{R_1} \rangle$ and $b_2 = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), ac_{L_2}, ac_{R_2} \rangle$ and a graph E with a pair of total, injective, and jointly surjective morphisms $(e_R : R_1 \hookrightarrow E, e_L : L_2 \hookrightarrow E)$, a* target/source pattern *over (b_1, b_2) is a pair of partial application conditions over R_1 and L_2 of the form $(\exists(e_R, ac_E), \exists(e_L, ac_E))$ with ac_E being a composed partial negative application condition over E.*

A pair of morphisms with the same codomain $(m'_1 : R_1 \hookrightarrow G, m_2 : L_2 \hookrightarrow G)$ satisfies a target/source pattern (tar, src), denoted $(m'_1, m_2) \models (tar, src)$, if m'_1 and m_2 satisfy tar and src via a common injective morphism, i.e. if there exists $y : E \hookrightarrow G$ with $y \models ac_E$, $y \circ e_R = m'_1$, and $y \circ e_L = m_2$.

Again, the existential conditions' morphisms e_R and e_L are required to be total while ac_E is a composed partial negative application condition. Since, by Definition D.2, an s/t-pattern sequence is defined based on source patterns, target patterns, and target/source patterns, we do not need to adjust that definition.

Unfortunately, the transfer of a target pattern $tar = \exists(t, ac_T)$ over a rule b into a source pattern $src = \text{L}(b, tar)$ via the L-construction (as in steps SC$_1$-3 and SC$_\text{k}$-3 of the Seq-construction) does not work for target patterns with composed partial negative application conditions. Elements created by the rule – whose images are deleted upon application of the L-construction – affect the result of the Shift-construction. Simply transferring a composed partial negative application condition from a target to a source pattern fails to take this effect into account. Therefore, we require at least a partial expansion – i.e. execution of Shift – that transfers the composed negative application condition to a reduced context of the rule; more specifically, to a context that contains all nodes and edges modified by the rule and all nodes adjacent to modified edges.

To determine this context, we use the notion of *reduced rules*. Reduced rules are graph rules that contain at least the elements (and adjacent nodes) modified by the graph rules they are derived from. A reduced rule provides sufficient information to correctly apply the original rule once its applicability via a match has been ensured. In itself, a reduced rule carries less information than the original rule because most elements not modified by the orignial rule can be omitted.

Definition 7.25 (reduced rule [1])**.** *Given a plain rule $b = (L \hookleftarrow K \hookrightarrow R)$, a* reduced rule *of b is any rule $b^* = (L^* \hookleftarrow K^* \hookrightarrow R^*)$ with injective morphisms $r^+ : R^* \hookrightarrow R$, $l^+ : L^* \hookrightarrow L$ and $k^+ : K^* \hookrightarrow K$ such that for all graphs G, H and injective morphisms $m : L \hookrightarrow G$ and $m' : R \hookrightarrow H$ we have $G \Rightarrow_{b,m,m'} H \Leftrightarrow G \Rightarrow_{b^*,m \circ l^+, m' \circ r^+} H$.*

$$\begin{tikzcd}
L \arrow[dd, bend right, "m"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[dd, bend left] & R \arrow[dd, bend left, "m'"] \\
L^* \arrow[u, hook, "l^+"'] & K^* \arrow[l, hook', "l^*"] \arrow[r, hook, "r^*"'] \arrow[u, hook', "k^+"] & R^* \arrow[u, hook', "r^+"] \\
G & D \arrow[l, hook'] \arrow[r, hook] & H
\end{tikzcd}$$

The equivalence $G \Rightarrow_{b,m,m'} H \Leftrightarrow G \Rightarrow_{b^*,m \circ l^+, m' \circ r^+} H$ describes that a reduced rule carries all the information necessary to correctly apply the rule for a given match once its applicability has been guaranteed. Given the knowledge that a rule application of b to a graph G via a match m is possible, actually applying the rule only requires information about elements to be deleted and created by the rule. Elements preserved by the rule application (i.e. elements in K) are mostly required to determine applicability, which, by definition, was already established. They are also required if their image under r connects to created elements, in which case they have to be included in the reduced rule. Hence, a reduced rule may omit a number of elements in K and their images under l and r from its specification.

The construction of reduced rules implements this idea as follows:

Construction 7.26 (construction of reduced rules [2])**.** *Given a plain rule $b = (L \hookleftarrow K \hookrightarrow R)$, a reduced rule $b^* = (L^* \hookleftarrow K^* \hookrightarrow R^*)$ of b can be constructed as follows:*

1. *Create an injective morphism $k^+ : K^* \hookrightarrow K$ with K^* a subgraph of K that contains at least all nodes n that fulfill at least one of the following two conditions:*
   a) *The image of n under r is adjacent to an edge in $R \setminus r(K)$.*
   b) *The image of n under l is adjacent to an edge in $L \setminus l(K)$.*
2. *Create $K^* \overset{r^*}{\hookrightarrow} R^* \overset{r^+}{\hookrightarrow} R$ as the pushout complement of k^+ and r.*
3. *Create $K^* \overset{l^*}{\hookrightarrow} L^* \overset{r^+}{\hookrightarrow} L$ as the pushout complement of k^+ and l.*

Proof. We have to show that for all graphs G, H and injective morphisms $m : L \hookrightarrow G$ and $m' : R \hookrightarrow H$, it holds that $G \Rightarrow_{b,m,m'} H \Leftrightarrow G \Rightarrow_{b^*,m \circ l^+, m' \circ r^+} H$. Both pushout complements (1) and (2) exist by construction of K^*.

$$\begin{tikzcd}
L \arrow[dd, bend right=40, "m"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[dd, hook, bend left=20, "k"] & R \arrow[dd, bend left=40, "m'"] \\
L^* \arrow[u, hook, "l^+"] \arrow[ur, phantom, "(1)"] & K^* \arrow[l, hook', "l^*"] \arrow[r, hook, "r^*"'] \arrow[u, hook, "k^+"'] \arrow[ur, phantom, "(2)"] & R^* \arrow[u, hook, "r^+"'] \\
G & D \arrow[l, hook', "g"'] \arrow[r, hook, "h"] & H
\end{tikzcd}$$

Only if. Since (1) is a pushout by construction and (G, g, m) is a pushout by precondition, $(1) + (G, g, m)$ is a pushout by pushout composition. Since (2) is a pushout by construction and (H, h, m') is a pushout by precondition, $(2) + (H, h, m')$ is a pushout by pushout composition. Then, we have $G \Rightarrow_{b^*, m \circ l^+, m' \circ r^+} H$.

If. Since $(G, g, m \circ l^+)$ – i.e. $(1) + (G, g, m)$ – is a pushout by precondition and (1) is a pushout by construction, (G, g, m) is a pushout by pushout decomposition. Since $(H, h, m' \circ r^+)$ – i.e. $(2) + (H, h, m')$ – is a pushout by precondition and (2) is a pushout by construction, (H, h, m') is a pushout by pushout decomposition. Then, we have $G \Rightarrow_{b, m, m'} H$, concluding the proof. □

Note that this construction does not usually construct a unique reduced rule. In general, a rule may have a number of different reduced rules since the construction allows a choice between possible graphs K^* and morphisms k^+. In most cases, it makes sense to choose K^* as small as possible because the size of K^* determines the computational effort of the next step: converting composed negative application conditions into composed partial negative application conditions via a combination of Shift and PShift. The intent is to create a target pattern with a composed partial negative application condition where the condition covers at least the context of the reduced rule, but does not extend to the complete right rule side.

Example 7.27 (reduced rule)**.** Figure 7.14 shows the plain part $(L \hookleftarrow K \hookrightarrow R)$ of brake = $\langle (L \hookleftarrow K \hookrightarrow R), \text{true}, \text{true} \rangle$ and a corresponding reduced rule brake* = $(L^* \hookleftarrow K^* \hookrightarrow R^*)$. Since the rule's application conditions are trivial, we will also denote the rule's plain part as brake. K^* contains t1 and s1, whose images under $l \circ k^+$ are adjacent to a deleted edge (i.e. an edge in $L \setminus l(K)$) of type isAt. It also contains t2, whose image under $r \circ k^+$ is adjacent to a created edge (i.e. an edge in $R \setminus r(K)$) of type isAt. Here, K^+ only contains these elements, which are required of the construction. In general, K^* may contain other elements from K, although the intention usually is to keep K^* as small as possible.

R^* and L^* are then created as the respective pushout complements to (1) and (2). Note that R^* indeed contains exactly those elements whose images under r^+ are modified by the rule (one edge of types isAt and brake each) or connected to an edge modified by the rule (nodes t1, t2, and s). The same applies for L^*.

The figure also shows a rule application $G \Rightarrow_{\mathsf{brake}, m, m'} G'$. By Definition 7.25 (p. 197), we know that given the rule's applicability via $m : L \hookrightarrow G$ and $m' : R \hookrightarrow G'$ application of $G \Rightarrow_{\mathsf{brake}^*, m \circ l^+, m' \circ r^+} G'$ yields the same result. Informally, we ensure applicability of brake via m to G; then, for the actual application, we only need to know which elements are subject to change. That information is contained in the reduced rule: we delete an edge of type isAt between t1 and s, create an edge of the same type between t2 and s, and add an edge of type brake to the shuttle s. Elements without modifications – all remaining tracks and next edges – are not relevant for the rule's application once their existence has been established. △

7.3.2. Construction of Pattern Sequences

With reduced rules, we can properly transfer composed partial negative application conditions from target to source patterns via a combination of Shift, PShift, and L. We integrate the respective steps into a refined Seq-construction:

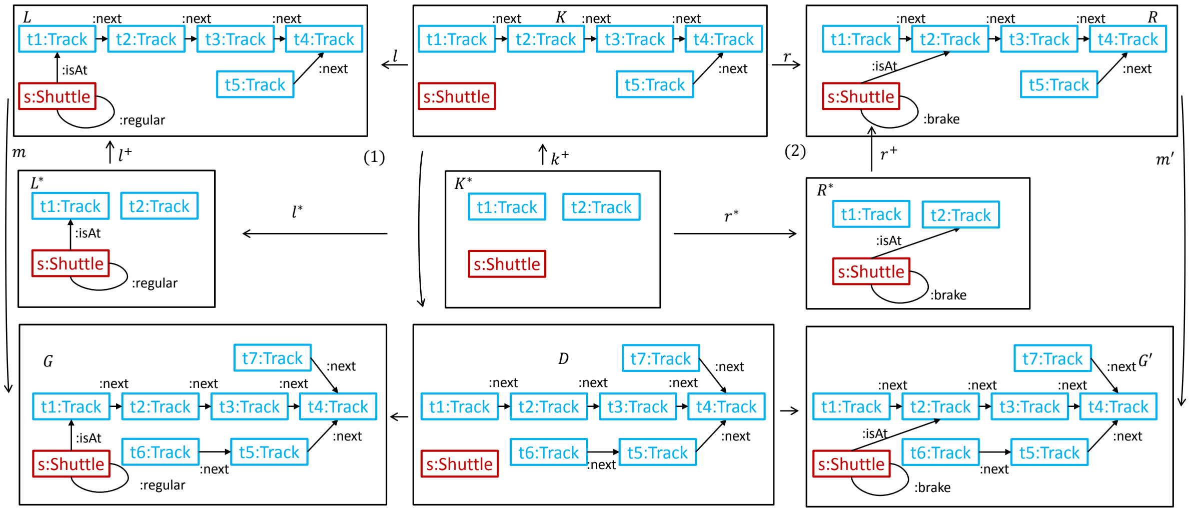


Figure 7.14. – Reduced rule $\mathsf{brake}^* = (L^* \hookleftarrow K^* \hookrightarrow R^*)$ of rule $\mathsf{brake} = (L \hookleftarrow K \hookrightarrow R)$ and rule applications $G \Rightarrow_{\mathsf{brake},m,m'} G'$ and $G \Rightarrow_{\mathsf{brake}^*,m\circ l^+,m'\circ r^+} G'$

Theorem 7.28 (construction of s/t-pattern sequences)**.** *There is a construction* $\mathrm{Seq}_k^{r,p}$ *such that for every graph pattern* $C = \exists(i_P, ac_P)$*, rule set* $\mathcal{R}$*, and* $k \geq 1$*,* $\mathrm{Seq}_k^{r,p}(\mathcal{R}, C)$ *is a set of* k*-sequences of* s/t*-patterns such that:*

1. *For each transformation sequence* $trans$ *to* $\mathcal{R}$ *and of length* k *leading to* C*, there exists a* $seq \in \mathrm{Seq}_k^{r,p}(\mathcal{R}, C)$ *such that* $trans \vDash seq$*.*
2. *Given a* $seq \in \mathrm{Seq}_k^{r,p}(\mathcal{R}, C)$*, for every transformation sequence* $trans = G_0 \Rightarrow_{b_1,m_1,m_1'} \ldots \Rightarrow_{b_k,m_k,m_k'} G_k$ *with* $trans \vDash seq$*,* $trans$ *leads to* C*.*

Construction. $\mathrm{Seq}_k^{r,p}$ *is inductively constructed as follows (with appropriate indexes and index sets), starting with* $\mathrm{Seq}_1^{r,p}$*, which consists of five steps* SC_1*-1 to* SC_1*-5:*

SC_1-1: For each rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$ *and a correspondingly chosen reduced rule* $b^* = (L^* \hookleftarrow K^* \hookrightarrow R^*)$ *with, in particular, a morphism* $r^+ : R^* \hookrightarrow R$*, we have* $\mathrm{Shift}(i_{R^*}, \exists i_P) = \bigvee_{i\in I} \exists t_i^*$ *and* $\mathrm{Shift}(r^+, \mathrm{Shift}(i_{R^*}, \exists i_P)) = \bigvee_{j\in J} \exists t_j$ *as shown below. Then,* $\bigvee_{j\in J_b} tar_{b,j}^p$ *with* $tar_{b,j}^p = \exists(t_j, \mathrm{PShift}(t_j', \mathrm{Shift}(t_i'^*, ac_P)))$ *is a disjunction of target patterns. We denote its fragments as* $ac_{T_i^*} = \mathrm{Shift}(t_i'^*, ac_P)$ *and* $ac_{T_j}^p = \mathrm{PShift}(t_j', ac_{T_i^*})$*.*

$$\begin{tikzcd}
 & & \varnothing \arrow[r, hook, "i_P"] \arrow[d, hook, "i_{R^*}"] & P \triangleleft ac_P \arrow[d, hook, "t_i'^*"'] \\
L^* \arrow[d, hook, "l^+"] & K^* \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] \arrow[d, hook, "k^*"] & R^* \arrow[r, hook, "t_i^*"'] \arrow[d, hook, "r^+"] & T_i^* \triangleleft ac_{T_i^*}\text{=Shift}(t_i'^*, ac_P) \arrow[d, hook, "t_j'"'] \\
L & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[r, hook, "t_j"'] & T_j \triangleleft ac_{T_j}^p\text{=PShift}(t_j', ac_{T_i^*})
\end{tikzcd}$$

SC_1-2: For each such target pattern $tar_{b,j}^p = \exists(t_j, \mathrm{PShift}(t_j', ac_{T_i^*}))$ *thusly constructed and the corresponding reduced rule* b^**, we have* $\mathrm{L}(b, \exists t_j) = \mathrm{false}$ *or* $\mathrm{L}(b, \exists t_j) = \exists s_j$*. For the latter case shown below,* $src_{b,j}'^p = \exists(s_j, ac_{S_j}'^p)$ *is a source pattern over* L *where* $ac_{S_j}'^p = \mathrm{PShift}(s_j^+, \mathrm{L}(b', ac_{T_i^*}))$*, with* $b' = (S_i^* \hookleftarrow K' \hookrightarrow T_i^*)$ *the rule constructed via* $\mathrm{L}(b^*, \exists(t_j' \circ t_i^*)) = \exists(s_j^+ \circ s_i^*)$ *(leading to the pushouts (1) and (2)).*

$$\begin{tikzcd}
 & ac_L \triangleright L \arrow[dddl, bend right, "s_j"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[dddr, bend left, "t_j"] & \\
 & L^* \arrow[u, hook, "l^+"] \arrow[d, hook, "s_i^*"] \arrow[dr, phantom, "(2)"] & K^* \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] \arrow[u, hook, "k^+"] \arrow[d, hook, "k'"] \arrow[dr, phantom, "(1)"] & R^* \arrow[u, hook, "r^+"'] \arrow[d, hook, "t_i^*"'] & \\
 & ac_{S_i^*}{=}\mathrm{L}(b',ac_{T_i^*}) \triangleright S_i^* \arrow[dl, hook, "s_j^+"] & K' \arrow[l, hook', "l'"] \arrow[r, hook, "r'"'] & T_i^* \triangleleft ac_{T_i^*} \arrow[dr, hook, "t'_j"] & \\
ac'^p_{S_j}{=}\mathrm{PShift}(s_j^+,ac_{S_j^*}) \triangleright S_j & & K'' \arrow[ll, hook', "l''"] \arrow[rr, hook, "r''"'] & & T_j \triangleleft ac^p_{T_j}{=}\mathrm{PShift}(t'_j,ac_{T_i^*})
\end{tikzcd}$$

SC$_1$-3: For the case $src'^p_{b,j} = \exists(s_j, ac'^p_{S_j})$ illustrated above, $src^p_{b,j} = \exists(s_j, ac^p_{S_j})$ with $ac^p_{S_j} = ac'^p_{S_j} \wedge$ $\mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b))$ is a source pattern.

SC$_1$-4: For each such pair of source and target pattern $src^p_{b,j}$ and $tar^p_{b,j}$, $src^p_{b,j} \Rightarrow_b tar^p_{b,j}$ is a 1-sequence of s/t-patterns.

SC$_1$-5: Finally, we define $\mathrm{Seq}^{r,p}_1(\mathcal{R}, C) = \{src^p_{b,j} \Rightarrow_b tar^p_{b,j} \mid b \in \mathcal{R} \wedge j \in J_b\}$ as the set of these sequences.

Given $\mathrm{Seq}^{r,p}_k(\mathcal{R}, C)$, we construct $\mathrm{Seq}^{r,p}_{k+1}(\mathcal{R}, C)$ as follows.

SC$_k$-1: For each sequence $seq = src_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} tar_k \in \mathrm{Seq}^{r,p}_k(\mathcal{R}, C)$ with $src^p_1 = \exists(s_1 : L_1 \hookrightarrow S_1, ac^p_{S_1})$ and $ac^p_{S_1} = \mathrm{PShift}(s^+_1, ac_{S^*_1}) \wedge \mathrm{Shift}(s_1, ac_{L_1})$, rule $b = \langle(L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true}\rangle \in \mathcal{R}$ and a correspondingly chosen reduced rule $b^* = (L^* \hookleftarrow K^* \hookrightarrow R^*)$ with, in particular, a morphism $r^+ : R^* \hookrightarrow R$, we have $\mathrm{Shift}(r^+, \mathrm{Shift}(i_{R^*}, \exists i_{S_1})) = \bigvee_{j \in J} \exists t_j$ as shown below. Then, $\bigvee_{j \in J} tar^p_j$ with $tar^p_j = \exists(t_j, \mathrm{PShift}(t'_j, \mathrm{Shift}(s'^*_i, \mathrm{Shift}(s^+_1, ac_{S^*_1}) \wedge \mathrm{Shift}(s_1, ac_{L_1}))))$ is a disjunction of target patterns over R. We denote its application conditions as $ac_{T^*_i} = \mathrm{Shift}(s'^*_i, \mathrm{Shift}(s^+_1, ac_{S^*_1}) \wedge \mathrm{Shift}(s_1, ac_{L_1}))$ and $ac^p_{T_j} = \mathrm{PShift}(t'_j, ac_{T^*_i})$; then, $tar_{b,j} = \exists(t_j, ac^p_{T_j})$.

$$\begin{tikzcd}
 & & & \varnothing \arrow[dl, hook', "i_{R^*}"'] \arrow[r, hook, "i_{L_1}"'] & L_1 \triangleleft ac_{L_1} \arrow[dd, bend left, hook, "s_1"] \\
L^* \arrow[d, hook, "l^+"] & K^* \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] \arrow[d, hook, "k^*"] & R^* \arrow[d, hook, "r^+"'] \arrow[dr, hook, "t_i^*"] & & S_1^* \triangleleft ac_{S_1^*} \arrow[d, hook, "s_1^+"'] \\
L & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[dr, hook, "t_j"'] & T_i^* \arrow[d, hook, "t'_j"'] & S_1 \triangleleft ac_{S_1}{=}\mathrm{PShift}(s_1^+,ac_{S_1^*}){\wedge}\mathrm{Shift}(s_1,ac_{L_1}) \arrow[l, hook', "s'^*_i"'] \arrow[dl, hook', "s'_j"] \\
 & & & T_j \triangleleft ac^p_{T_j}{=}\mathrm{PShift}(t'_j,ac_{T_i^*}) &
\end{tikzcd}$$

SC$_k$-1$^+$: For each such target pattern $tar^p_{b,j}$, $src^{p+}_{1,j} = \exists(s'_j \circ s_1, ac^p_{T_j})$ is a source pattern over L_1 and $(tar^p_{b,j}, src^{p+}_{1,j})$ is a target/source pattern over (b, b_1).

SC$_1$-2: For each such target pattern $tar^p_{b,j} = \exists(t_j, \mathrm{PShift}(t'_j, ac_{T^*_i}))$ thusly constructed and the corresponding reduced rule b^*, we have $\mathrm{L}(b, \exists t_j) = \mathrm{false}$ or $\mathrm{L}(b, \exists t_j) = \exists s_j$. For the latter case shown below, $src'^p_{b,j} = \exists(s_j, ac'^p_{S_j})$ is a source pattern over L where $ac'^p_{S_j} = \mathrm{PShift}(s^+_j, \mathrm{L}(b', ac_{T^*_i}))$, with $b' = (S^*_i \hookleftarrow K' \hookrightarrow T^*_i)$ being the rule constructed via $\mathrm{L}(b^*, \exists(t'_j \circ t^*_i)) = \exists(s^+_j \circ s^*_i)$ (leading to the pushouts (1) and (2)).

$$\begin{tikzcd}
 & ac_L \rhd L \arrow[ddd, bend right, "s_j"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[ddd, bend left, "t_j"] & \\
 & L^* \arrow[u, hook', "l^+"'] \arrow[d, hook, "s_i^*"] \arrow[dr, phantom, "(2)"] & K^* \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] \arrow[u, hook', "k^+"'] \arrow[d, hook, "k'"] \arrow[dr, phantom, "(1)"] & R^* \arrow[u, hook', "r^+"] \arrow[d, hook, "t_i^*"'] & \\
 & ac_{S_i^*}\text{=L}(b',ac_{T_i^*}) \rhd S_i^* \arrow[dl, hook, "s_j^+"] & K' \arrow[l, hook', "l'"] \arrow[r, hook, "r'"'] & T_i^* \lhd ac_{T_i^*} \arrow[dr, hook, "t_j'"'] & \\
ac_{S_j}'^{p}\text{=PShift}(s_j^+,ac_{S_j^*}) \rhd S_j & & K'' \arrow[ll, hook', "l''"] \arrow[rr, hook, "r''"'] & & T_j \lhd ac_{T_j}^{p}\text{=PShift}(t_j',ac_{T_i^*})
\end{tikzcd}$$

SC

SC_1-3: For the case $src_{b,j}'^{p} = \exists(s_j, ac_{S_j}'^{p})$ illustrated above, $src_{b,j}^{p} = \exists(s_j, ac_{S_j}^{p})$ with $ac_{S_j}^{p} = ac_{S_j}'^{p} \wedge$ $\text{Shift}(s_j, ac_L \wedge \text{Appl}(b))$ *is a source pattern.*

SC_k-4: For each such pair of source and target pattern $src_{b,j}^{p}$ and $tar_{b,j}^{p}$, $src_{b,j}^{p} \Rightarrow_b tar_{b,j}^{p}, src_1^{p+} \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k^{p}$ is a k+1-sequence of s/t-patterns.

SC_k-5: Finally, we define $\text{Seq}_{k+1}^{r,p}(\mathcal{R}, C) = \{src_{b,j}^{p} \Rightarrow_b tar_{b,j}^{p}, src_{1,j}^{p+} \Rightarrow \dots \Rightarrow tar_k^{p} \mid b \in \mathcal{R} \wedge j \in J_b \wedge seq \in \text{Seq}_k^{r,p}(\mathcal{R}, C)\}$ as the set of these sequences.

Also, given a set of rules $\mathcal{R}$ and a composed graph pattern $\mathcal{C} = \bigwedge_{i\in I} \neg C_i$ with graph patterns C_i, we define $\text{Seq}_k^{r,p}(\mathcal{R}, \neg\mathcal{C}) = \bigcup_{i\in I} \text{Seq}_k(\mathcal{R}, C_i)$ and $\text{SEQ}_k^{r,p}(\mathcal{R}, \mathcal{C}) = \bigcup_{1\leq j\leq k} \text{Seq}_j^{r,p}(\mathcal{R}, \mathcal{C})$.

Proof. In order to prove the correctness of the construction, we will show its equivalence to the Seq-construction established in Theorem T.1r (p. 130). In particular, given an arbitrary rule set $\mathcal{R}$, $k \geq 1$, and a pattern C, we will show that for each s/t-pattern sequence $seq^* \in \text{Seq}_k^{r}(\mathcal{R}, C)$, there is an s/t-pattern sequence $seq \in \text{Seq}_k^{r,p}(\mathcal{R}, C)$ such that $seq \equiv seq^*$ and vice versa. We will prove this by showing the equivalence of the construction steps for an arbitrary rule b:

SC_1-1: Given $\bigvee_{j\in J} tar_{b,j} = \text{Shift}(i_R, C)$ and $\bigvee_{j\in J} tar_{b,j}^{p}$ as above, for all $j \in J$, we have $tar_{b,j} \equiv tar_{b,j}^{p}$.

SC_1-2: Given target patterns $tar_{b,j}$ and $tar_{b,j}^{p}$ as above and $src_{b,j}'$ and $src_{b,j}'^{p}$ as constructed by step SC_1-2 of the respective constructions, we have $src_{b,j}' \equiv src_{b,j}'^{p}$.

SC_1-3: Given source patterns $src_{b,j}'$ and $src_{b,j}'^{p}$ as above and $src_{b,j}$ and $src_{b,j}^{p}$ as constructed by step SC_1-3 of the respective constructions, we have $src_{b,j} \equiv src_{b,j}^{p}$.

SC_1-1. Since, in general, $\text{Shift}(a \circ b, ac) = \text{Shift}(a, \text{Shift}(b, ac))$ [GHE14], we have

$$\begin{aligned} tar_{b,j} &= \exists(t_j, ac_{T_j}) = \exists(t_j, \text{Shift}(t_j' \circ t_i'^{*}, ac_P)) = \exists(t_j, \text{Shift}(t_j', \text{Shift}(t_i'^{*}, ac_P))) \\ &= \exists(t_j, \text{Shift}(t_j', ac_{T_i^*})) \end{aligned}$$

as depicted below. By Lemma 7.20 (p. 194), we have $tar_{b,j} \equiv \exists(t_j, \text{PShift}(t_j', ac_{T_i}^{*}))$, which is equal to $tar_{b,j}^{p}$.

$$\begin{tikzcd}
 & & \varnothing \arrow[r, hook, "i_P"] \arrow[d, hook, "i_{R^*}"] & P \lhd ac_P \arrow[d, hook, "t_i'^{*}"'] \\
L^* \arrow[d, hook, "l^+"] & K^* \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] \arrow[d, hook, "k^*"] & R^* \arrow[r, hook, "t_i^*"'] \arrow[d, hook, "r^+"] & T_i^* \lhd ac_{T_i^*}\text{=Shift}(t_i'^{*},ac_P) \arrow[d, hook, "t_j'"'] \\
L & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[r, hook, "t_j"'] & T_j \lhd ac_{T_j}^{p}\text{=PShift}(t_j',ac_{T_i^*})
\end{tikzcd}$$

SC₁-2. Given $tar_{b,j} = \exists(t_j, ac_{T_j})$ and the corresponding source pattern $src'_{b,j} = \exists(s_j, ac'_{S_j})$ with $ac'_{S_j} = \mathrm{L}(b'', ac_{T_j})$ with $b'' = (S_j \leftarrow K'' \hookrightarrow T_j)$ as depicted below and b'' created via $\mathrm{L}(b, \exists t_j) = \exists s_j$. Furthermore, we have $tar^p_{b,j} = \exists(t_j, \mathrm{PShift}(t'_j, ac_{T^*_i}))$ and the corresponding source pattern $src'^p_{b,j} = \exists(s_j, ac'^p_{S_j})$ with $ac'^p_{S_j} = \mathrm{PShift}(s^+_j, \mathrm{L}(b', ac_{T^*_i})))$ and $b' = (S^*_j \leftarrow K' \hookrightarrow T^*_i)$ – and with b' created via $\mathrm{L}(b, \exists(t'_j \circ t^*_i)) = \exists(s^+_j \circ s^*_i)$, leading to the pushouts (1), (2), (3), and (4).

We need to show $ac'_{S_j} \equiv ac'^p_{S_j}$, i.e. $\mathrm{L}(b'', ac_{T_j}) \equiv \mathrm{PShift}(s^+_j, \mathrm{L}(b', ac^p_{T^*_i}))$. We have

$$\begin{aligned}
\mathrm{PShift}(s^+_j, \mathrm{L}(b', ac_{T^*_i})) &\equiv \mathrm{Shift}(s^+_j, \mathrm{L}(b', ac_{T^*_i})) && \text{(by Lemma 7.20 (p. 194))}\\
&\equiv \mathrm{L}(b'', \mathrm{Shift}(t'_j, ac_{T^*_i})) && \text{(since (3) and (4) are pushouts [GHE14])}\\
&= \mathrm{L}(b'', ac_{T_j}). && \text{(see above, SC}_1\text{-1)}
\end{aligned}$$

$$\begin{tikzcd}
 & ac_L \triangleright L \arrow[dddl, bend right, "s_j"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[dddr, bend left, "t_j"] & \\
 & L^* \arrow[u, hook, "l^+"'] \arrow[d, hook, "s^*_j"] \arrow[dr, phantom, "(2)"] & K^* \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] \arrow[u, hook, "k^+"'] \arrow[d, hook, "k'"] \arrow[dr, phantom, "(1)"] & R^* \arrow[u, hook, "r^+"] \arrow[d, hook, "t^*_i"'] & \\
 & ac_{S^*_j} = \mathrm{L}(b', ac_{T^*_i}) \triangleright S^*_i \arrow[dl, hook, "s^+_j"] \arrow[dr, phantom, "(4)"] & K' \arrow[l, hook', "l'"] \arrow[r, hook, "r'"'] \arrow[d, hook, "k''"'] & T^*_i \triangleleft ac_{T^*_i} \arrow[dr, hook, "t'_j"'] \arrow[dl, phantom, "(3)"] & \\
ac'^p_{S_j} = \mathrm{PShift}(s^+_j, ac_{S^*_j}) \triangleright S_j & & K'' \arrow[ll, hook', "l''"] \arrow[rr, hook, "r''"'] & & T_j \triangleleft ac^p_{T_j}
\end{tikzcd}$$

SC₁-3. Since, as shown above, $ac'_{S_j} \equiv ac'^p_{S_j}$, and furthermore, $src_{b,j} = \exists(s_j, ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b)))$ and $src^p_{b,j} = \exists(s_j, ac'^p_{S_j} \wedge \mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b)))$, we have $src_j \equiv src^p_j$.

The required equivalences hold analogously for steps SC$_\mathrm{k}$-1, SC$_\mathrm{k}$-1$^+$, SC$_\mathrm{k}$-2, and SC$_\mathrm{k}$-3. □

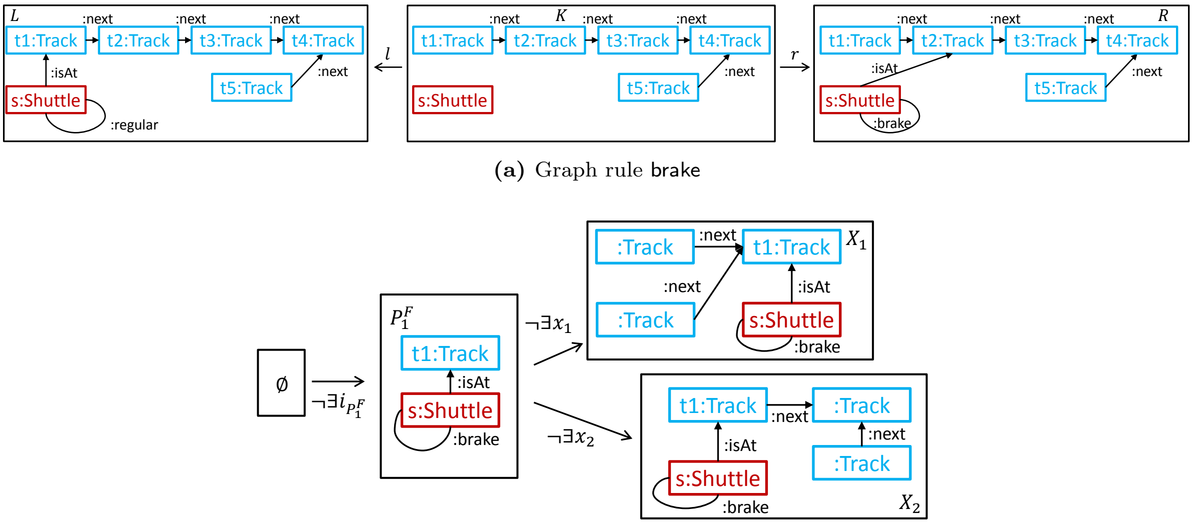
As explained above, the main idea of this version of the Seq-construction is to avoid computational effort in transferring composed negative application conditions via Shift, while we still transfer the (positive) existential condition of a pattern – or, in step SC$_\mathrm{k}$-1, a sequence's leftmost source pattern. However, integrating left application conditions and applicability conditions of rules into the s/t-pattern sequences as partial application conditions would be significantly more involved. Hence, these are regularly transferred to composed total application conditions by $\mathrm{Shift}(s_j, ac_L \wedge \mathrm{Appl}(b))$ in steps SC₁-3 and SC$_\mathrm{k}$-3, respectively.

Example 7.29 (Seq-construction with composed partial negative application conditions)**.** Figure 7.15(a) shows the rule brake already discussed earlier; Figure 7.15(b) shows the composed forbidden pattern $\mathcal{F} = \neg F_1$ again. Here, we shortly discuss the result of $\mathrm{Seq}^{r,p}_1(\mathcal{R}, \neg\mathcal{F}) = \mathrm{Seq}^{r,p}_1(\mathcal{R}, F_1)$ with $\mathcal{R} = \{\mathsf{brake}\}$.

Figure 7.16 shows the details and construction of a target pattern $tar = \exists(t, ac^p_T)$ by step SC₁-1; tar is only one of several target patterns created by this step. It depicts only the right sides of $\mathsf{brake} = \langle(L \leftarrow K \hookrightarrow R), \mathrm{true}, \mathrm{true}\rangle$ and a corresponding reduced rule $\mathsf{brake}^* = (L^* \leftarrow K^* \hookrightarrow R^*)$.

Then, T is one possible overlapping of R and P^F_1 via t and $t' \circ t'^*$. T^* is one of the overlappings of R^* and P^F_1 via t^* and t'^*. The composed (total) application condition $\neg\exists x_{21} \wedge \neg\exists x_{22} \wedge \neg\exists x_{23}$ is the result of $\mathrm{Shift}(t'^*, \neg\exists x_2)$ – i.e. of shifting the first of the pattern's negative application conditions to the context of T^*. We get a similar result for $\neg\exists x_2$. Then,

$$ac_{T^*} = \mathrm{Shift}(t'^*, \neg\exists x_1 \wedge \neg\exists x_2) = \bigwedge_{1 \le i \le 3} \neg\exists x_{1i} \wedge \bigwedge_{1 \le i \le 3} \neg\exists x_{2i}$$



(a) Graph rule brake

(b) Composed forbidden pattern $\mathcal{F} = \neg F_1 = \neg\exists(i_{P_1^F}, \neg\exists x_1 \wedge \neg\exists x_2)$

Figure 7.15. – Graph rule brake and forbidden pattern F_1

results in a composed negative application condition consisting of only six individual negative application conditions. From that, $ac_T^p = \mathrm{PShift}(t', ac_{T^*})$ (not depicted) creates a composed partial negative application condition. In particular,

$$ac_T^p = \mathrm{PShift}(t', ac_{T^*}) = \bigwedge_{1\leq i\leq 3} \neg\exists\left\langle t', x_{1i}\right\rangle \wedge \bigwedge_{1\leq i\leq 3} \neg\exists\left\langle t', x_{2i}\right\rangle.$$

Finally, $tar = \exists(t, ac_T^p)$ is our target pattern. It describes situations where, after application of brake, a switch is two tracks ahead ($\exists t$) while neither the current track ($\bigwedge_{1\leq i\leq 3} \neg\exists\left\langle t', x_{1i}\right\rangle$) nor its successor ($\bigwedge_{1\leq i\leq 3} \neg\exists\left\langle t', x_{2i}\right\rangle$) are switches. If we had not used partial application conditions, we would have needed to compute $\mathrm{Shift}(t' \circ t'^*, ac_P)$, which would result in a significantly greater number of negative application conditions (cf. Example 7.16 (p. 189)).

Figure 7.17 shows the s/t-pattern sequence seq created from tar by the remaining steps. We have $seq \in \mathrm{Seq}_1^{r,p}(\mathcal{R}, F_1)$ and

$$seq = \exists(s, ac_S^p) \Rightarrow_{\mathsf{brake}} \exists(t, ac_T^p).$$

The composed (total) application condition $ac_{S^*} = \mathrm{L}(b', ac_{T^*})$ (with $b' = (S^* \hookleftarrow K' \hookrightarrow T^*)$) is not shown in detail. Similar to ac_{T^*} in step SC$_1$-1, $\mathrm{PShift}(s^+, ac_{S^*})$ creates a composed partial negative application condition ac_S^p; then, $src = \exists(s, ac_{S^*}^p)$. △

Note that step SC$_\mathrm{k}$-1 expands composed partial negative application conditions in source patterns to composed total negative application conditions: the total fragment of $ac_{S_1}^p = \mathrm{PShift}(s_1^+, ac_{S_i^*})$ is subjected to the Shift-construction in $\mathrm{Shift}(s_1^+, ac_{S_i^*})$. The goal is to have both this condition and $\mathrm{Shift}(s_1, ac_{L_1})$ – the transferred left application condition of the sequence's leftmost rule (b_1) – defined as total application conditions over the same graph. Then, both are shifted to the context of overlappings T_i^* of the reduced rule in question (R^*, part of b^*) and the leftmost source pattern S_1: $ac_{T_i^*} = \mathrm{Shift}(s_i'^*, \ldots)$. From there, $ac_{T_i^*}$ is again transformed to a composed partial negative application condition $ac_{T_j}^p = \mathrm{PShift}(t_j', ac_{T_i^*})$ in the new target pattern $tar_{b,j}^p = \exists(t_j, ac_{T_j}^p)$.

Alternatively, we could have shifted $ac_{S_i^*}$ to overlappings of R^* and S_1^* (instead of S_1). Since S_1^* is a subgraph of S_1, execution would be faster. Then, however, we might not be able

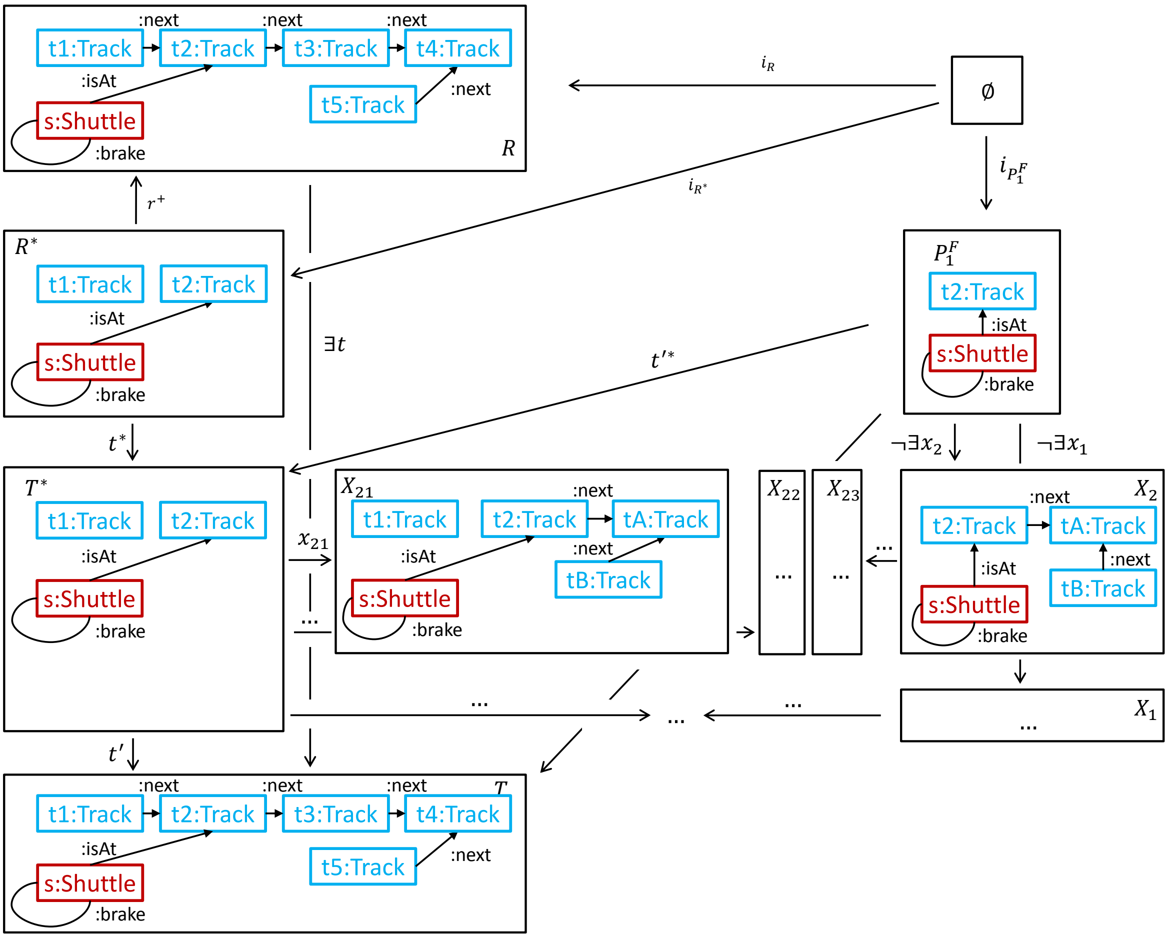


Figure 7.16. – Target pattern $tar = \exists(t, \mathrm{PShift}(t', \mathrm{Shift}(t'^*, ac_P)))$ created by step SC$_1$-1, with $ac_P = \neg\exists x_1 \wedge \neg\exists x_2$

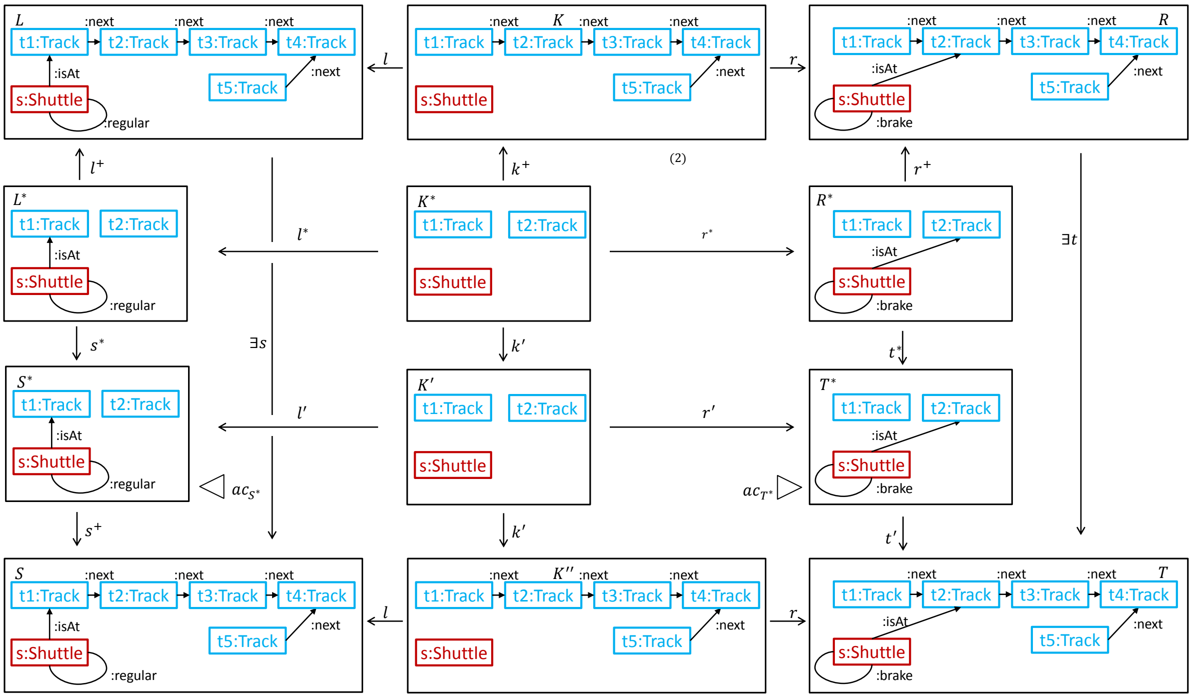


Figure 7.17. – S/t-pattern sequence $seq = \exists(s, ac^p_S) \Rightarrow_{\text{brake}} \exists(t, ac^p_T)$

to transfer $\text{Shift}(s_1, ac_{L_1})$ such that both application conditions are defined over the same context. As a result, subsequent construction steps would become more involved. Exploring this alternative may be desirable if composed negative application conditions in patterns have a greater impact than left application conditions in rules. The approach here, however, follows the first idea.

7.3.3. Analysis of Pattern Sequences

Of course, using partial application conditions only to expand them later only makes sense if s/t-pattern sequences can be discarded as potential counterexamples in between – or if we perform verification for $k = 1$. The implementation (Section 6.7) of the restricted approach's version of the Seq-construction (Theorem T.1r (p. 130)) interweaves the analysis of sequences (Theorems T.2r (p. 143) and T.3r (p. 149)) with their construction: until length k is reached, sequences are analyzed after each iteration. In particular, the approach checks whether reduced source and target patterns imply individual guaranteed patterns, forbidden patterns, or start configuration patterns, depending on whether we perform k-inductive invariant checking or k–1-bounded backwards model checking. If we choose $k = 1$, we also need to perform these checks.

However, we cannot apply Theorem 6.8 (p. 120) to check implication of patterns – it requires composeed (total) negative application conditions. We have to extend the theorem to allow patterns with composed partial negative application conditions. The extension shown here is based on a previous incarnation of the theorem [1].

Theorem 7.30 (implication of patterns)**.** *Let $C = \exists(i_P : \varnothing \hookrightarrow P, ac)$ be a pattern with a composed total negative application condition $ac = \bigwedge_{i \in I} \neg\exists(x_i : P \hookrightarrow X_i)$. Furthermore, let $C' = \exists(i_{P'} : \varnothing \hookrightarrow P', ac'^p \wedge ac^*)$ be a pattern with a composed partial negative application condition $ac'^p = \bigwedge_{j \in J} \neg\exists(x'_j : P' \hookleftarrow\!\!\rightharpoonup X'_j)$ (where $x'_j = \langle q', q_j \rangle$, $q : Q \hookrightarrow P'$, and $q_j : Q_j \hookrightarrow X'_j$) and a composed total negative application condition $ac^* = \bigwedge_{u \in U} \neg\exists(x^*_u : P' \hookrightarrow X^*_u)$.*

$$\begin{tikzcd}
P \arrow[d, hook, "x_i"] & \varnothing \arrow[l, hook', "i_P"'] \arrow[r, hook, "i_{P'}"] & P' \arrow[dl, hook', "x_u^*"'] \arrow[d, hook, "x_j'"'] & Q \arrow[l, hook', "q'"'] \arrow[dl, hook', "q_j"] \\
X_i & X_u^* & X_j' &
\end{tikzcd}$$

We have $C' \vDash C$, if one of the following conditions is fulfilled:

1'-A. There is a u in U such that x_u^* is bijective.

1'-B. There is a j in J such that there are injective and jointly surjective morphisms $a : P' \hookrightarrow A$ and $b : X_j' \hookrightarrow A$ with $a \circ q' = b \circ q_j$ and a is bijective (see diagram below).

$$\begin{tikzcd}
 & P' \arrow[dl, hook', "a"'] \arrow[d, hook, "x_j'"'] & Q \arrow[l, hook', "q'"] \arrow[dl, hook', "q_j"] \\
A & X_j' \arrow[l, hook', "b"] &
\end{tikzcd}$$

1. *There exists an injective morphism $m : P \hookrightarrow P'$ such that, for each i in I, one of the following conditions is fulfilled:*

   1. *There exists a $j \in J$ with $q'(Q) \subseteq m(P)$ and an injective morphism $y : X_j' \hookrightarrow X_i$ such that $y \circ q_j = x_i \circ m'$, with $m' = m^{-1} \circ q'$.*

$$\begin{tikzcd}
P \arrow[r, hook, "m"'] \arrow[d, hook, "x_i"'] & P' \arrow[l, shift right, "m^{-1}"'] \arrow[d, hook, "\langle q', q_j \rangle = x_j'"'] & Q \arrow[l, hook', "q'"] \arrow[dl, hook', "q_j"] \arrow[ll, bend right, "m'"'] \\
X_i & X_j' \arrow[l, hook', "y"] &
\end{tikzcd}$$

   2. *With (Q', a, b) the pullback over m and q' and (P'', a', b') the pushout over a and b, and with $\mathrm{Shift}(b', \neg\exists x_i) = \bigwedge_{k \in K} \neg\exists x_k''$, for each k in K, there is a j in J and an injective morphism $y : X_j' \hookrightarrow X_k''$ such that $y \circ q_j = x_k'' \circ a'$.*

$$\begin{tikzcd}
 & Q' \arrow[dl, hook', "a"'] \arrow[dr, hook, "b"] & \\
P \arrow[r, hook, "b'"] \arrow[dr, hook, "m"'] \arrow[dd, hook, "x_i"'] & P'' \arrow[d, hook] & Q \arrow[l, hook', "a'"] \arrow[dl, hook', "q'"] \arrow[dd, hook, "q_j"] \arrow[ll, bend right, "m'"'] \\
 & P' \arrow[d, hook, "x_k''"'] \arrow[dr, hook, "x_j'"] & \\
X_i \arrow[r, hook, "n_k"'] & X_k'' & X_j' \arrow[l, hook', "y"]
\end{tikzcd}$$

   3. *With $\mathrm{Shift}(m, \neg\exists x_i) = \bigwedge_{k \in K} \neg\exists x_k''$, for each k in K, there is a u in U and an injective morphism $y : X_u^* \hookrightarrow X_k''$ such that $y \circ x_u^* = x_k''$ (diagram below).*

$$\begin{tikzcd}
P \arrow[r, hook, "m"'] \arrow[d, hook, "x_i"] & P' \arrow[d, hook, "x''_k"'] \arrow[dr, hook, "x^*_u"] \arrow[dr, phantom, "="'] & \\
X_i \arrow[r, hook, "n_k"'] & X''_k & X^*_u \arrow[l, hook', "y"]
\end{tikzcd}$$

Proof. We have to show $\forall G(G \vDash C' \Rightarrow G \vDash C)$.

By precondition, case (1'-A), there is a u in U such that x^*_u is bijective. Consider an arbitrary graph G with $i_G \vDash \exists i_{P'}$, i.e. there is an injective morphism $g' : P' \hookrightarrow G$ and $g \circ i_{P'} = i_G$. Since x^*_u (for the specific u) is bijective, there is an injective morphism $y : X^*_u \hookrightarrow G$ such that $y \circ x^*_u = g'$. Hence, we have $g' \nvDash \neg\exists x^*_u$, C' is equivalent to false, and $C' \vDash C$ holds trivially.

By precondition, case (1'-B), there is a j in J such that there are injective and jointly surjective morphisms $a : P' \hookrightarrow A$ and $b : X'_j \hookrightarrow A$ with $a \circ q' = b \circ q_j$ and a bijective. Consider an arbitrary graph G with $i_G \vDash \exists i_{P'}$, i.e. there is an injective morphism $g' : P' \hookrightarrow G$ and $g \circ i_{P'} = i_G$. Then, we have $g \circ a^{-1} \circ b : X'_j \hookrightarrow G$ and, with $g' \circ a^{-1} \circ b \circ q_j = g' \circ a^{-1} \circ a \circ q' = g' \circ q'$, we have $g' \nvDash ac_{P'}$. Hence, C' is equivalent to false and $C' \vDash C$ holds trivially.

$$\begin{tikzcd}
& P' \arrow[dl, hook, shift right, "a"'] \arrow[d, hook, "x'_j"'] \arrow[dd, hook, bend left, "g'"] & Q \arrow[l, hook', "q'"] \arrow[dl, hook, "q_j"] \\
A \arrow[ur, hook, shift right, "a^{-1}"'] \arrow[dr, hook] & X'_j \arrow[l, hook', "b"] & \\
& G &
\end{tikzcd}$$

For case (1), consider an arbitrary graph G with $G \vDash C'$. By definition of satisfaction, we have $i_G \vDash C'$, implying the existence of an injective morphism $g' : P' \hookrightarrow G$ with $g' \vDash ac' \wedge ac^*$. By precondition (B), there is an injective morphism $m : P \hookrightarrow P'$ and hence, we have an injective morphism $g : P \to G$ with $g = g' \circ m$.

We will show $g \vDash ac$ by contradiction. Suppose $g \nvDash ac$, implying the existence of a $x = x_i$ for some $i \in I$ and a corresponding injective morphism $c : X \hookrightarrow G$ with $g = c \circ x$, i.e. $g \nvDash \neg\exists x$. By assumption, one of the following is true:

1. There is an injective morphism $y : X' \hookrightarrow X$ with $X' = X'_j$ and $x' = x'_j = \langle q', q\rangle$ for some $j \in J$ such that $q'(Q) \subseteq m(P)$ and $y \circ q = x \circ m'$ with $m' = m^{-1} \circ q'$. Consequently, there is an injective morphism $y' : X' \hookrightarrow G$ with $y' = c \circ y$. In addition, we have:

$$\begin{tikzcd}
P \arrow[r, hook, shift right, "m"'] \arrow[d, hook, "x"'] \arrow[ddr, hook, bend left, "g"] & P' \arrow[l, hook', shift right, "m^{-1}"'] \arrow[d, hook, "\langle q',q\rangle=x'"'] \arrow[dd, hook, bend left, "g'"] & Q \arrow[l, hook', "q'"] \arrow[dl, hook, "q"] \arrow[ll, hook', bend right, "m'"'] \\
X \arrow[dr, hook, "c"'] & X' \arrow[l, hook', "y"] \arrow[d, hook, "y'"'] & \\
& G &
\end{tikzcd}$$

$$\begin{aligned}
& & y \circ q &= & x \circ m' & & \\
&\Longrightarrow & c \circ y \circ q &= & c \circ x \circ m' & & \\
&\Longrightarrow & y' \circ q &= & g \circ m' & & (y' = c \circ y \text{ and } c \circ x = g) \\
&\Longrightarrow & y' \circ q &= & g' \circ m \circ m' & & (g = g' \circ m) \\
&\Longrightarrow & y' \circ q &= & g' \circ m \circ m^{-1} \circ q' & & (m' = m^{-1} \circ q') \\
&\Longrightarrow & y' \circ q &= & g' \circ q' & & (q'(Q) \subseteq m(P))
\end{aligned}$$

This implies $g' \not\models \neg\exists x'$ and therefore $g' \not\models ac'$, which is a contradiction. Thus, this case leads to $g \models ac$ and, with $g : P \hookrightarrow G$, to $G \models C$.

2. Given (Q', a, b) as the pullback over m and q' and (P'', a', b') as the pushout over a and b, we have $\text{Shift}(b', \neg\exists x) = \bigwedge_{k \in K} \neg\exists x''_k$ and, by assumption, for each k in K, there is a j in J and an injective morphism $y : X'_j \hookrightarrow X''_k$ such that $y \circ q_j = x''_k \circ a'$.

$$\begin{tikzcd}
 & Q' \arrow[dl, hook', "a"'] \arrow[dr, hook, "b"] & \\
P \arrow[r, hook, "b'"] \arrow[dr, hook, "m"'] \arrow[dd, hook, "x"'] \arrow[dddr, hook, "g"'] & P'' \arrow[d, hook, "p"] \arrow[dd, hook, bend right, "x''_k"'] & Q \arrow[l, hook', "a'"'] \arrow[ll, hook', bend right, "m'"'] \arrow[dl, hook', "q'"] \arrow[dd, hook, "q_j"] \\
 & P' \arrow[dr, hook, "x'_j"] \arrow[dd, hook, bend left, "g'"] & \\
X \arrow[r, hook, "n_k"'] \arrow[dr, hook, "c"'] & X''_k \arrow[d, hook, "c'"] & X'_j \arrow[l, hook', "y"] \arrow[dl, hook', "y'"] \\
 & G &
\end{tikzcd}$$

Furthermore, by construction and the pushout property, there is an injective morphism $p : P'' \hookrightarrow P'$ such that $p \circ a' = q'$, $p \circ b' = m$, and hence, $g' \circ p \circ b' = g$. Then, by the Shift-lemma, $g \not\models \exists x$ implies $g' \circ p \not\models \bigwedge_{k \in K} \neg\exists x''_k$ and, as a result, there is a k in K such that there is an injective morphism $c' : X''_k \hookrightarrow G$ and $c' \circ x''_k = g' \circ p$. Then, there is an injective morphism $y' : X'_j \hookrightarrow G$ with $y' = c' \circ y$.

In addition, we have:

$$\begin{aligned}
y' \circ q_j &= c' \circ y \circ q_j \\
&= c' \circ x''_k \circ a' \\
&= g' \circ p \circ a' \\
&= g' \circ q'
\end{aligned}$$

This implies $g' \not\models \neg\exists x'_j$ and therefore $g' \not\models ac'$, which is a contradiction. Thus, this case leads to $g \models ac$ and, with $g : P \hookrightarrow G$, to $G \models C$.

3. With $\text{Shift}(m, \neg\exists x) = \bigwedge_{k \in K} \neg\exists x''_k$, there is a u in U and an injective morphism $y : X^*_u \hookrightarrow X''_k$ such that $y \circ x^*_u = x''_k$.

$$\begin{tikzcd}
P \arrow[r, hook, "m"'] \arrow[d, hook, "x"'] \arrow[ddr, bend right, hook, "g"] & P' \arrow[d, hook, "x''_k"] \arrow[dr, hook, "x^*_u"] \arrow[dr, phantom, "=", near end] \arrow[dd, bend right=40, hook, "g'"'] & \\
X \arrow[r, hook, "n_k"'] \arrow[dr, hook, "c"'] & X''_k \arrow[d, hook, "c'"] & X^*_u \arrow[l, hook', "y"'] \arrow[dl, hook', "y'"] \\
& G &
\end{tikzcd}$$

Since $g' \circ g = m$ and with the Shift-lemma, we have $g' \nvDash \bigwedge_{k \in K} \neg\exists x''_k$, implying the existence of an injective morphism $c' : X''_k \hookrightarrow G$ with $c' \circ x''_k = g'$ for some k in K. Then, by assumption, there is an injective morphism $y : X^*_u \hookrightarrow X''_k$ with $y \circ x^*_u = x''_k$ for some u in U, implying the existence of an injective morphism $y' : X^*_u \hookrightarrow G$ with $y' = c' \circ y$. Also, we have $y' \circ x^*_u = c' \circ y \circ x^*_u = c' \circ x''_k = g'$.

This implies $g' \nvDash \neg\exists x^*_u$ and therefore $g' \nvDash ac'$, which is a contradiction. Thus, this case leads to $g \vDash ac$ and, with $g : P \hookrightarrow G$, to $G \vDash C$.

□

This approach to implication of patterns relates to the original Theorem 6.8 (p. 120) as follows:

Case (1'-A) is similar to case 1' of Theorem 6.8 (p. 120): with a bijective morphism in a negated existential condition, the pattern is equivalent to false and implication holds trivially.

Case (1'-B) similarly checks equivalence to false for the partial part of the implying pattern's application condition. Here, we cannot simply check the morphisms for bijectivity. Instead, looking for morphism pairs with the specified properties amounts to checking whether the equivalent composed total negative application condition containts a bijective morphism. However, finding relevant morphism pairs or showing their absence does not require complete expansion of the composed partial negative application conditions to its total equivalent (via Shift).

Case (1) and its subcase work are similar to cases (1) and (2) of Theorem 6.8 (p. 120). If the implying pattern's existential condition's context (P') contains the other pattern's context P as a subgraph, C' implies C if the implied pattern's negative application conditions (ac) can be matched by the implying pattern's composed negative application conditions. In order to match an individual (total) negative application condition, there are three possibilities:

1. The implying pattern's composed partial negative application condition ac'^p is defined (totally) over a graph Q (and partially over P'). If the image of Q in P' is a subgraph of the mapping $m : P \hookrightarrow P'$ of the implied pattern's context, we can attempt to directly map negative application conditions in ac'^p to negative application conditions in ac. If successful, this is a significant advantage over the alternatives below and the original approach in case (2) of Theorem 6.8 (p. 120): we do not have to transfer ac to the context of P' by computing Shift(m, ac).
2. If the image of Q in P' is not a subgraph of $m(P)$, we cannot compare conditions in ac to partial negative application conditions in ac'^p. The default procedure would be to transfer ac to the context of P' (via Shift(m, ac)). However, there is a more efficient option: the pullback and pushout construction described in the theorem allows us to transfer ac to a graph P'' via b' such that $b'(P)$ does contain the image of Q as a subgraph. Then, negative application conditions in Shift(b', ac) can be compared to

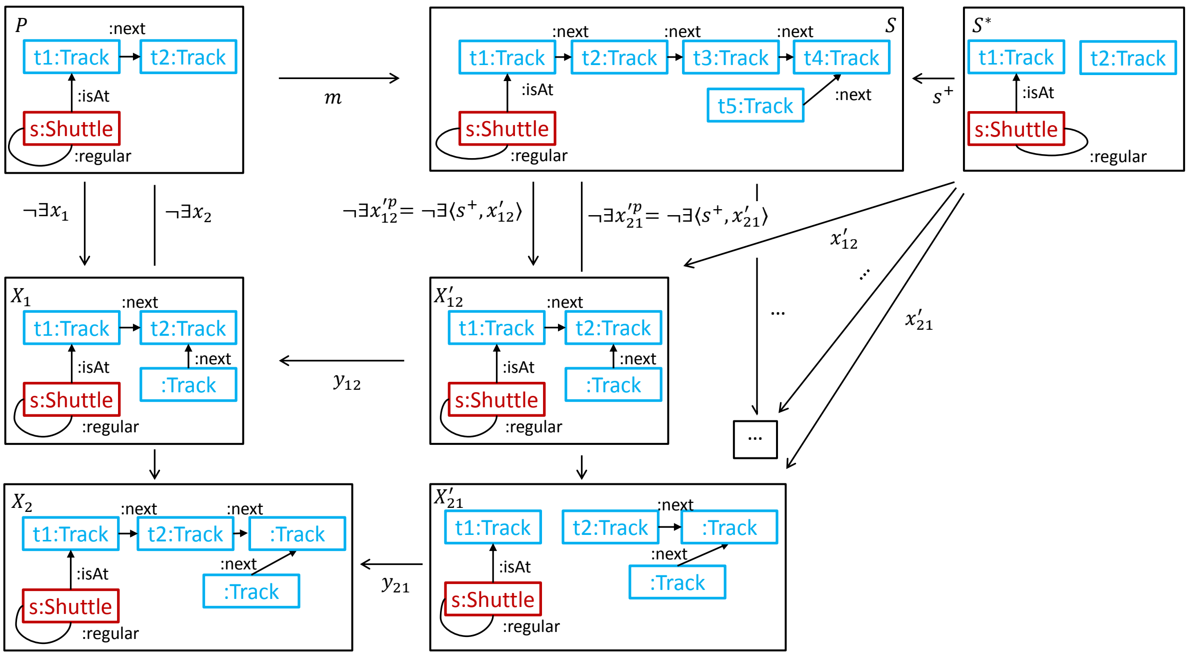


Figure 7.18. – Reduced source pattern $src_{|\varnothing} = \exists(i_S, ac^p_S)$ with $ac^p_S = \neg\exists\,\langle s^+, x'_{12}\rangle \wedge \cdots \wedge \neg\exists\,\langle s^+, x'_{21}\rangle \wedge \ldots$ and implied pattern $C = \exists(i_P, \neg\exists x_1 \wedge \neg\exists x_2)$

partial negative application conditions in ac'^p. Usually, computing $\mathrm{Shift}(b', ac)$ requires less effort than executing $\mathrm{Shift}(m, ac)$ because P'' is a subgraph of P'. Likewise, we do not have to expand the composed partial negative application condition ac'^p to its total equivalent over P'.

3. Lastly, if both options above fail, we can attempt to map negative application conditions in ac to the total part of the implying pattern's composed negative application condition (ac^*). This part exists because the Seq-construction transfers left application condition's of graph rules to the source pattern as total composed negative application conditions, not in a partial fashion. Hence, this case applies the same principle as case (2) of the original Theorem 6.8 (p. 120).

Example 7.31 (implication of patterns)**.** Consider the patterns shown in Figure 7.18. On the right side, we have the reduced source pattern (cf. Example 7.29 (p. 202))

$$src_{|\varnothing} = \exists(i_S, ac^p_S) \text{ with } ac^p_S = \neg\exists x'^p_{12} \wedge \cdots \wedge \neg\exists x'^p_{21} \wedge \cdots = \neg\exists\left\langle s^+, x'_{12}\right\rangle \wedge \cdots \wedge \neg\exists\left\langle s^+, x'_{21}\right\rangle \wedge \ldots .$$

In particular, $\neg\exists x'_{12}$ is the result of $\mathrm{L}((S^* \hookleftarrow K' \hookrightarrow T^*), \neg\exists x_{12})$ as per step SC$_1$-2 of the Seq-construction (again, see Example 7.29 (p. 202)) – and $\neg\exists x'_{21}$ is the result of $\mathrm{L}((S^* \hookleftarrow K' \hookrightarrow T^*), \neg\exists x_{21})$. The existential part $\exists(i_S, \ldots)$ of the pattern describes a situation where a chain of four tracks has a switch (t4) at its end and a shuttle in speed mode regular on the first track. Its partial composed negative application condition (ac^p_S, only fragments depicted) specifies that neither t2 nor any of its successors is a switch. Note that the latter condition applies to any successor, not just to t3: $\neg\exists x'^p_{21}$ is a partial negative application condition over S and its interface S^* does not specify a determined successor for t2.

Now, suppose we have the pattern $C = \exists(i_P, \neg\exists x_1 \wedge \neg\exists x_2)$ depicted on the left side of Figure 7.18. With rule priorities (Section 7.2), this could be the rule applicability constraint of a higher-priority rule meant to preempt the execution of the brake rule – unless a switch is one $(\neg\exists x_1)$ or two tracks $(\neg\exists x_2)$ ahead. Then, if $src_{|\varnothing}$ implies C, $src \Rightarrow_{\mathsf{brake}} tar$ could be discarded as a counterexample.

Since $src_{|\varnothing}$ is not equivalent to false, we first need to find an injective morphism $m : P \hookrightarrow S$. There is exactly one such morphism, which is also depicted in Figure 7.18. Checking pattern implication via Theorem 6.8 (p. 120) requires composed total negative application conditions in patterns – we would have to compute $\text{Shift}(m, \neg\exists x_1 \wedge \neg\exists x_2)$ and $\text{Shift}(s^+, \neg\exists x'_{12} \wedge \dots \wedge \neg\exists x'_{21} \wedge \dots)$. Theorem 7.30 (p. 205) offers a more efficient alternative: we can easily find injective morphisms $y_{12} : X'_{12} \hookrightarrow X_1$ and $y_{21} : X'_{21} \hookrightarrow X_2$ fulfilling the conditions of case (1.1). Note that $s^+(S^*) \subseteq m(P)$ – we do not even need to apply the partial expansion of case (1.2). The result: $src_{|\varnothing}$ indeed implies C. All graphs satisfying the reduced source pattern also satisfy C.

A more intuitive view is that both $\neg\exists x_1 \wedge \neg\exists x_2$ and $\neg\exists x'_{12} \wedge \dots \wedge \neg\exists x'_{21} \wedge \dots$ are defined over the same graph – or rather, they are defined over graphs P and S^* whose images under m and s^+ in S are identical. The condition $\neg\exists x_1 \wedge \neg\exists x_2$ expresses that, given two subsequent tracks `t1` and `t2` (and a shuttle), neither `t2` nor its successor are switches. The condition $\neg\exists x'_{12} \wedge \neg\exists x'_{21}$ in the context of S^* expresses a nearly identical situation. Then, it makes sense that the respective conditions also correspond to each other when considered in the context of S. In S, the tracks `t1` and `t2` (and the shuttle) are part of a longer chain of tracks with a switch at the end. Note that the switch is not close enough to the shuttle to be a contradiction to $\neg\exists x'^{p}_{12} \wedge \neg\exists x'^{p}_{21}$, which would make $src_{|\varnothing}$ equivalent to false and trivially imply C. △

Now that we have a means to compare patterns where the implying patterns has a composed partial negative application condition, we can apply Theorem T.2r (p. 143) with little change. Given a set of rules $\mathcal{R}$ and a composed pattern $\mathcal{F}$ and $\mathcal{H}$ as usual, we replace $\text{Seq}^r_k(\mathcal{R}, \neg\mathcal{F})$ by $\text{Seq}^{r,p}_k(\mathcal{R}, \neg\mathcal{F})$ – and use Theorem 7.30 instead of Theorem 6.8 (p. 120) to check implication of patterns. Likewise, an updated version of Theorem T.3r (p. 149) uses $\text{SEQ}^{r,p}_{k-1}(\mathcal{R}, \neg\mathcal{F})$ instead of $\text{SEQ}^r_{k-1}(\mathcal{R}, \neg\mathcal{F})$.

Theorem T.2e-pn (k-inductive invariant checking with partial negative application conditions)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ and $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$ be a composed forbidden pattern and composed guaranteed pattern, respectively.*

$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ if, for all sequences $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ with $seq \in \text{Seq}^{r,p}_k(\mathcal{R}, \neg\mathcal{F})$, one of the following conditions holds:

1. $\exists z, v(1 \leq z \leq k \wedge (src_{z|\varnothing} \vDash H_v \vee src_{z|\varnothing} \vDash F_v))$.
2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.

Proof. This follows from the proof of Theorem T.2r (p. 143), the equivalence of $\text{Seq}^r_k(\mathcal{R}, C)$ and $\text{Seq}^{r,p}_k(\mathcal{R}, C)$ shown in Theorem 7.28 (p. 199), and Theorem 7.30 (p. 205). □

Theorem T.3e-pn (k−1-bounded backward model checking with partial negative application conditions)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o \in O} \neg S_o$ be a composed forbidden pattern, composed guaranteed pattern, and composed start configuration pattern, respectively, with $\mathcal{S} \vDash \mathcal{F}$.*

For all graphs $G \in \text{REACH}_{k-1}(GG, \mathcal{H})$ and graph grammars $GG = (GTS, G_0)$ with $GG \in \text{IND}(GTS, \mathcal{S})$, we have $G \vDash \mathcal{F}$, if for all sequences $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n$ with $seq \in \text{SEQ}^{r,p}_{k-1}(\mathcal{R}, \neg\mathcal{F})$, one of the following conditions holds:

1. $\exists z, v(1 \leq z \leq n \wedge (src_{z|\varnothing} \vDash H_v))$.
2. $\exists v(tar_{k|\varnothing} \vDash H_v)$.
3. $\exists v(src_{1|\varnothing} \vDash SC_v)$.

Proof. This follows from the proof of Theorem T.3r (p. 149), the equivalence of $\text{Seq}^r_k(\mathcal{R}, C)$ and $\text{Seq}^{r,p}_k(\mathcal{R}, C)$ shown in Theorem 7.28 (p. 199), and Theorem 7.30 (p. 205). □

The effect of partial negative application conditions on performance is discussed in more detail in Chapter 9.

7.3.4. Implementation

Algorithms 7.7 and 7.8 show the implementation of the Seq-construction with composed partial negative application conditions; it follows Theorem 7.28 very closely. As before, the analysis of constructed s/t-pattern sequences for violations of forbidden and guaranteed patterns is interwoven in the construction. The implementation of Theorem 7.30 (p. 205) – checking implication of patterns – is significantly more involved in comparison to Theorem 6.8 (p. 120). However, this mostly concerns the number of cases to consider, not the type of constructions used; hence, we omit its details here.

```
Algorithm 7.7: createSequences($\mathcal{R}$, F)
  desc.  : implements $\mathrm{Seq}_1^{r,p}(\mathcal{R}, F)$ with optimizations and partial application
           conditions
  input  : a set $\mathcal{R}$ of graph rules, a forbidden pattern $F = \exists(i_P, ac_P)$
  output: the result of $\mathrm{Seq}_1(\mathcal{R}, F)$ minus sequences discarded for violations of $\mathcal{H}$ or $\mathcal{F}$

results ← ∅
foreach $b \in \mathcal{R}$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ and the
  corresponding reduced rule $b^* = (L^* \hookleftarrow K^* \hookrightarrow R^*)$ do
    foreach tar in $\bigvee_{j \in J} tar_j = \mathrm{Shift}(r^+, \mathrm{Shift}(i_{R^*}, \exists i_P))$
      with $tar = \exists t$ do                                                /* SC_1-1 */
        if not discardPattern($tar_{|\varnothing}, \mathcal{H}$) then           /* analysis */
            $ac_{T^*} = \mathrm{Shift}(t', ac_P)$                                   /* SC_1-1 */
            $tar \leftarrow \exists(t, \mathrm{PShift}(t', ac_{T^*}))$            /* SC_1-1 */
            $src' \leftarrow \mathrm{L}(b, \exists t)$                              /* SC_1-2 */
            if $src'$ has the form $\exists s$ then
                $b' \leftarrow (S^* \hookleftarrow K' \hookrightarrow T^*)$
                $ac'_S \leftarrow \mathrm{PShift}(s^+, \mathrm{L}(b', ac_{T^*}))$
                $src' = \exists(s, ac'_S)$                                          /* SC_1-2 */
                $src = \exists(s, ac'_S \wedge \mathrm{Shift}(s, ac_L \wedge \mathrm{Appl}(b)))$   /* SC_1-3 */
                if not discardPattern($src_{|\varnothing}, \mathcal{F} \cup \mathcal{H}$) then   /* analysis */
                    $results \leftarrow results \cup \{src \Rightarrow_b tar\}$          /* SC_1-4 */
return results                                                                  /* SC_1-5 */
```

Algorithm 7.8: extendSequences($\mathcal{R}$, *sequences*)

desc. : implements $\mathrm{Seq}^{r,p}_{i+1}(\mathcal{R}, F)$ (based on the result of $\mathrm{Seq}_i(\mathcal{R}, F)$) with optimizations and using partial application conditions

input : a set of sequences *results* of the form $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ with $src_1 = \exists(s_1, ac_{S_1})$ that is the result of $\mathrm{Seq}_i(\mathcal{R}, F)$, a set $\mathcal{R}$ of graph rules and correspondingly chosen reduced rules, a set $\mathcal{F}$ of forbidden patterns, a set $\mathcal{H}$ of guaranteed patterns

output: $\mathrm{Seq}_{i+1}(\mathcal{R}, F)$ minus sequences discarded for violations of $\mathcal{H}$ or F

```
results ← ∅
foreach $seq \in results$ with $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ do
    foreach $b \in \mathcal{R}$ with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \mathrm{true} \rangle$ and the
      corresponding reduced rule $b^* = (L^* \hookleftarrow K^* \hookrightarrow R^*)$ do
        foreach $tar$ in $\bigvee_{j \in J} tar_j = \mathrm{Shift}(i_{R^+}, \mathrm{Shift}(i_{R^*} \exists i_{S_1}))$
          with $tar = \exists t$ do                                          /* $SC_k$-1 */
            if not discardPattern($tar_{|\varnothing}, \varnothing, \mathcal{H}$) then                  /* analysis */
                $ac_{T^*} = \mathrm{Shift}(s'^* \, \mathrm{Shift}(s_1, ac_{L_1}) \wedge \mathrm{Shift}(s_1^+, ac_{S_1^*}))$     /* $SC_1$-1 */
                $ac_T = \mathrm{PShift}(t', ac_{T^*})$                                 /* $SC_1$-1 */
                $tar \leftarrow \exists(t, ac_T)$                                       /* $SC_1$-1 */
                $src_1^+ \leftarrow \exists(s' \circ s_1, ac_T)$                                 /* $SC_k$-1$^+$ */
                $src' \leftarrow \mathrm{L}(b, \exists t)$                                        /* $SC_1$-2 */
                if $src'$ has the form $\exists s$ then
                    $b' \leftarrow (S^* \hookleftarrow K' \hookrightarrow T^*)$                             /* $SC_1$-2 */
                    $ac'_S \leftarrow \mathrm{PShift}(s^+, \mathrm{L}(b', ac_{T^*}))$                        /* $SC_1$-2 */
                    $src' = \exists(s, ac'_S)$                                     /* $SC_1$-2 */
                    $src' = \exists(s, ac'_S \wedge \mathrm{Shift}(s, ac_L \wedge \mathrm{Appl}(b)))$              /* $SC_1$-3 */
                    if not discardPattern($src_{|\varnothing}, \mathcal{F}, \mathcal{H}$) then          /* analysis */
                        $results \leftarrow results \cup$
                        $\{src \Rightarrow_b (tar, src_1^+) \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k\}$            /* $SC_1$-4 */
return results                                                        /* $SC_k$-5 */
```

7.4. Composed Graph Patterns and Implication

In Section 6.8, we have discussed interactions of graph patterns as one possible factor leading to false negatives. Theorems T.2r (p. 143), T.3r (p. 149), and its extended variants only compare reduced source and target patterns with patterns in a composed pattern $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ or $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$ on an individual basis. They do not take into account how graph patterns interact – even when a pattern $src_{|\varnothing}$ does not imply any F_i and H_i individually, it could still imply $\neg(\mathcal{F} \wedge \mathcal{H})$. This makes sense from a formal perspective, too: $\exists i \forall G (G \vDash src_{|\varnothing} \Rightarrow (G \vDash F_i \vee G \vDash H_i))$ is not equivalent to $\forall G (G \vDash src_{|\varnothing} \Rightarrow \exists i (G \vDash F_i \vee G \vDash H_i))$.

We can address this problem by extending our approach for the analysis of reduced source and target patterns. This requires extending implication of individual patterns to comparisons of graph patterns with composed graph patterns. Existing work [Pen08a, Pen08b, Pen09] has solved this questions in its most general form: implication of unrestricted graph constraints. However, the problem is undecidable in general [HP09]; only the fragment of combinations of simple existential (possibly negated) conditions without nesting has been shown to be decidable [Pen09].

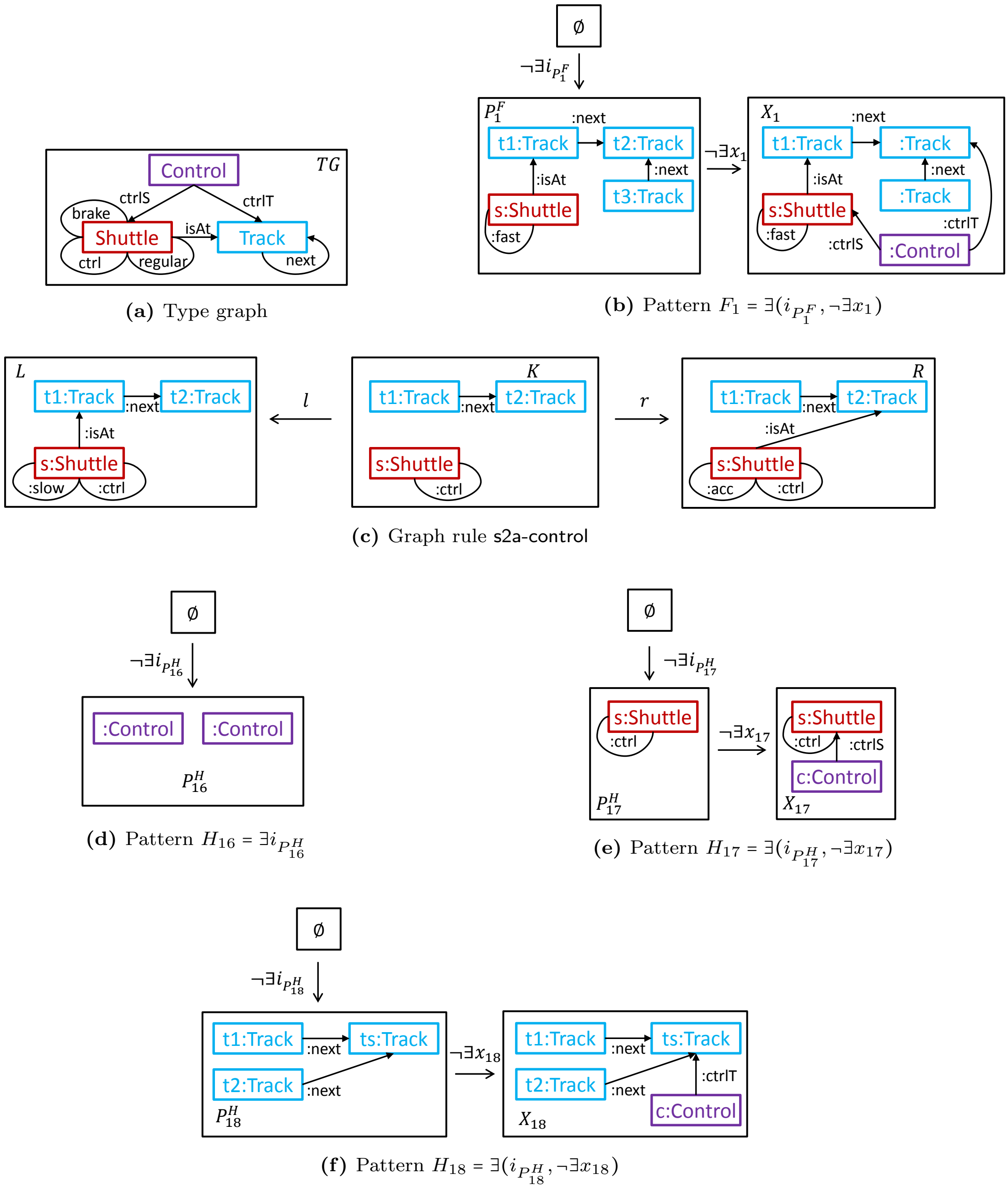
Here, we will not discuss a calculus solving the problem of implication of graph constraints; however, since we can focus on the fragment of graph patterns and composed graph patterns, we will describe the formal basis for an algorithm attempting to show whether a reduced source or target patterns indeed implies a violation of the corresponding composed forbidden and guaranteed patterns. Attempting is the key word here – the algorithm may not terminate, possible violating **Appl.-termination** unless forced to terminate by thresholds on (e.g.) runtime. If a system is considered unsafe unless the procedure terminates, **Appl.-soundness** can still be maintained. In some cases *[1]*, using this extension will reduce the number of false negatives and work towards **Appl.-deg.completeness**, which usually is the prime incentive. The effects on performance (**Appl.-performance**) vary depending on the scenario.

This extension does not require any extension of the formal model or the Seq-construction; it focuses on the analysis of source and target patterns. Before we delve deeper into that part, we will discuss an example highlighting the need for an extended notion of pattern implication.

Example 7.32 (implication of composed graph patterns)**.** Consider the fragments of an example system shown in Figure 7.19. There is an extended type graph (Figure 7.19(a)): a Control unit can connect some shuttles with tracks – specifically, switches, although this is not visible in the type graph – and shuttles can be annotated with ctrl, marking them as shuttles able to connect to a control unit. The intent of a control unit is to secure passage over a switch. Figure 7.19(b) shows an updated version of our safety property $\mathcal{F} = \neg F_1 = \neg\exists(i_{P_1^F}, \neg\exists x_1)$: a shuttle should not drive fast on a track ($\neg\exists(i_{P_1^F}, \dots)$) unless connected to the switch via a control unit ($\neg\exists x_1$).

Given this additional precaution, we can add new rules f2f-control, a2f-control, s2a-control, with the latter depicted in Figure 7.19(c): if a shuttle is marked with a ctrl edge, it does not have to slow down (or remain in speed mode slow) when approaching a switch, meaning the respective rules do not need any negative application conditions. Rules s2s, s2a, a2f, f2f, a2b, f2b, b2s from our standard example (Example 6.1 (p. 111)) remain mostly the same; only s2a, a2f, and f2f are extended by a negative application condition requiring the absence of a ctrl edge on the shuttle. In summary, we have a graph transformation system $GTS = (TG, R)$ with $\mathcal{R} = \{\text{s2s}, \text{s2a}, \text{a2f}, \text{f2f}, \text{a2b}, \text{f2b}, \text{b2s}, \text{f2f-control}, \text{a2f-control}, \text{s2a-control}\})$.

Using a control unit leads to additional requirements for the system's track and shuttle topology, which are implemented as guaranteed patterns. The negation of graph pattern H_{16} in Figure 7.19(d) forbids the existence of more than one control element. The negation of H_{17} (Figure 7.19(e)) requires every shuttle with a ctrl edge to have a connection to a control



(a) Type graph

(b) Pattern $F_1 = \exists(i_{P_1^F}, \neg\exists x_1)$

(c) Graph rule s2a-control

(d) Pattern $H_{16} = \exists i_{P_{16}^H}$

(e) Pattern $H_{17} = \exists(i_{P_{17}^H}, \neg\exists x_{17})$

(f) Pattern $H_{18} = \exists(i_{P_{18}^H}, \neg\exists x_{18})$

Figure 7.19. – Composed forbidden pattern $\mathcal{F} = \neg F_1$, graph rule s2a-control and fragments of composed guaranteed pattern $\mathcal{H} = \cdots \wedge \neg H_{16} \wedge \neg H_{17} \wedge \neg H_{18}$

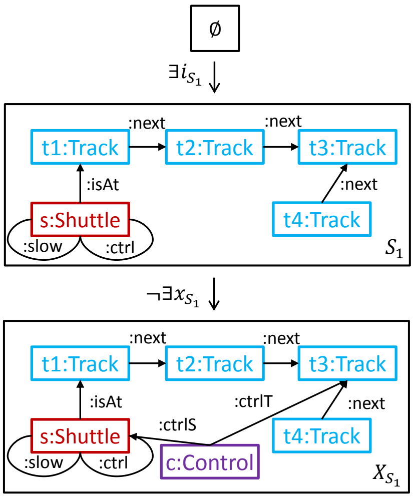
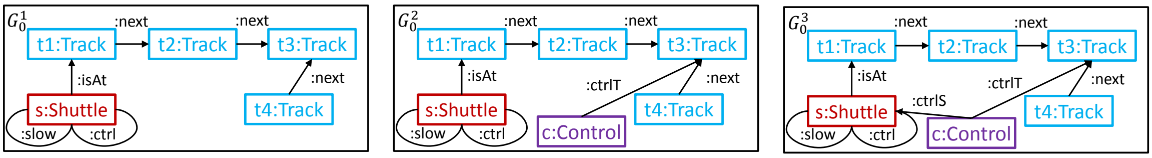


Figure 7.20. – Reduced source pattern $src_{1|\varnothing} = \exists(i_{S_1}, \neg\exists x_{S_1})$



(a) Graph G_0^1 with $G_0^1 \vDash src_{1|\varnothing}$, but $G_0^1 \nvDash \neg H_{18}$

(b) Graph G_0^2 with $G_0^2 \vDash src_{1|\varnothing}$ and $G_0^2 \vDash \neg\mathcal{H}_{18}$, but $G_0^2 \nvDash \neg H_{17}$

(c) Graph G_0^3 with $G_0^3 \vDash \neg H_{17} \wedge \neg H_{18}$, but $G_0^3 \nvDash src_{1|\varnothing}$

Figure 7.21. – Graphs G_0^1, G_0^2, and G_0^3 as candidates for satisfaction of $src_{|\varnothing}$

unit; note that $\neg(\exists i_{P_{17}^H}, \neg\exists x_{17})$ is equivalent to $\forall(i_{P_{17}^H}, \exists x_{17})$. Likewise, the negated of form of pattern H_{18} (Figure 7.19(f)) ensures that every switch is connected to a control unit. Together with the guaranteed patterns from Example 6.1 (p. 111), the guaranteed pattern's negations are part of a composed guaranteed pattern $\mathcal{H} = \bigwedge_{1 \leq j \leq 18} \neg H_j$.

Applying our usual procedure results in several counterexamples, with one reduced source pattern $src_{1|\varnothing} = \exists(i_{S_1}, \neg\exists x_{S_1})$ depicted in Figure 7.20. Its source pattern is part of an s/t-pattern sequence $seq = src_1 \Rightarrow_{\text{s2a-control}} (tar_1, src_2) \Rightarrow_{\text{a2f-control}} tar_2$ with $seq \in \text{Seq}_2^r(\mathcal{R}, F_1)$. The sequence represents a violation of $\neg F_1$ after application of s2a-control and a2f-control to graphs represented by $src_{1|\varnothing}$: a controlled shuttle in speed mode slow with a switch two tracks ahead $(\exists(i_{S_1}, \dots))$ and without a proper connection to the switch via the control unit $(\neg\exists x_{S_1})$. None of the examples forbidden of guaranteed patterns are implied by the reduced source pattern.

However, attempting to find a satisfying graph that also satisfies $\mathcal{F} \wedge \mathcal{H}$, which is required for the counterexample to be valid, will fail. An obvious candidate is G_0^1 in Figure 7.21(a) because it contains S_1, but not X_{S_1} and hence, $G_0^1 \vDash src_{1|\varnothing}$. However, it violates $\neg H_{18}$: the switch t3 is not connected to a control unit. G_0^2 (Figure 7.21(b)) does have a control unit c connected to the switch (and hence, $G_0^2 \vDash \neg\mathcal{H}_{18}$), but the shuttle lacks one despite its ctrl edge, which violates $\neg H_{17}$. Two separate control units are also forbidden $(\neg H_{16})$, which leaves us with G_0^3 in Figure 7.21(c) – here, we do have G_0^3 satisfies $\mathcal{H}$, but G_0^3 also contains X_{S_1}, meaning it does not satisfy $src_{1|\varnothing}$. In fact, all possible graphs will satisfy either $\mathcal{F} \wedge \mathcal{H}$ or $src_{1|\varnothing}$, but not both. Put differently, satisfaction of $src_{1|\varnothing}$ implies a violation of $\mathcal{F} \wedge \mathcal{H}$ and conversely, $\mathcal{F} \wedge \mathcal{H}$ implies $\neg src_{1|\varnothing}$, i.e. a violation of $src_{1|\varnothing}$. As a result, seq does not have any satisfying transformation sequences qualifying as counterexamples.

Indeed, all other remaining counterexamples in $\text{Seq}_2(\mathcal{R}, F_1)$, although not shown here, are false negatives – the system is safe. The extension discussed in this section is intended to discard spurious counterexamples in cases like this. △

7.4.1. Analysis of Pattern Sequences

In the examples above, we have inspected a counterexample by hand and can be reasonably sure we have found it to be a false negative; in general, we want this to be performed automatically. The proposed algorithm attempts to show implication of the negated reduced source or target pattern by the composed graph pattern $\mathcal{F} \wedge \mathcal{H}$. This is equivalent to showing implication of a violation of $\mathcal{F} \wedge \mathcal{H}$ by the reduced source (or target) pattern. Then, no graph can satisfy both the reduced source or target pattern and the composed forbidden and composed guaranteed graph patterns $\mathcal{F}$ and $\mathcal{H}$ – and the sequence of the pattern in question can be discarded. Similar to the approach in the example, the algorithm uses conditional forbidden patterns to generate context necessary to satisfy the composed graph patterns $\mathcal{F}$ and $\mathcal{H}$. This procedure is repeated until it has created elements forbidden by the reduced source (or target) pattern, which leads to the required contradiction of $\mathcal{F} \wedge \mathcal{H}$ and the reduced pattern.

More specifically, the algorithm starts with the existential part $\exists(i_P : \varnothing \hookrightarrow P)$ of a reduced source or target pattern $src_{|\varnothing} = \exists(i_P, ac_P)$. In order to show $\mathcal{F} \wedge \mathcal{H} \models \neg src_{|\varnothing}$ and discard it as a counterexample, the procedure applies *context generation* and *context reduction* in an alternating sequence. For this purpose, $\mathcal{F} \wedge \mathcal{H}$ are separated into two composed graph patterns $\mathcal{C}_1$ and $\mathcal{C}_2$: $\mathcal{C}_1$ combines (negated) patterns with non-trivial composed negative application conditions while $\mathcal{C}_2$ collects (negated) patterns describing a simple existential condition without further nesting, i.e. with true as their composed negative application condition. Since composed graph patterns are conjunctions of negated patterns, such a separation will always lead to the equality $\mathcal{F} \wedge \mathcal{H} = \mathcal{C}_1 \wedge \mathcal{C}_2$.

For context generation, the procedure uses graph patterns $C' = \exists(i_{P'}, ac_{P'})$ in a composed graph pattern ($\mathcal{C}_1$) to extend a pattern $C = \exists(i_P, ac_P)$ or, in the first step, $C = \exists i_P$. C' appears in its negated form $\neg\exists(i_{P'}, ac_{P'})$ in the composed graph pattern. Existence of P' implies violation of $ac_{P'}$, meaning that one of the negative application conditions' contexts has to be present. Hence, if P' can be found in S via a morphism $p : P' \hookrightarrow P$, we can extend $\exists i_P$ to $C^* = \exists(i_P, \mathrm{Shift}(p, ac_{P'}))$ in order to have its negation $\neg C^*$ implied by $\neg C'$.

After this first step, i.e. when extending a pattern $C = \exists(i_P, ac_P)$, the same idea is applicable if P' can be found in the context of a negative application condition $\exists(x : P \hookrightarrow X)$ in ac_P via an injective morphism $p : P' \hookrightarrow X$. Then, $\neg C \wedge \neg C'$ imply the negated result $\neg C^*$. In both cases, we required the connection via implication to establish a chain of implication relations that should eventually lead to $\mathcal{C}_1 \wedge \mathcal{C}_2 \models \neg src_{|\varnothing}$. Context generation is described by Lemma 7.33 below.

Context reduction, on the other hand, removes negative application conditions from a pattern's composed negative application condition. We consider a pattern $C = \exists(i_P, ac_P)$ and a composed graph pattern ($\mathcal{C}_2$) whose patterns are simple existential conditions with the trivial composed negative application condition true. Then, the result C^* of applying context reduction via $\mathcal{C}_2$ to C is a pattern $\exists(i_P, ac_P^*)$. The composed negative application condition ac_P^* is created by removing all negative application conditions $\neg\exists x : P \hookrightarrow X$ with $X \not\models \mathcal{C}$ from ac_P. Lemma 7.35 describes this procedure and establishes equivalence of $\mathcal{C}_2 \wedge \neg C$ and $\mathcal{C}_2 \wedge \neg C^*$. The forward direction of the equivalence is required to preserve the desired chain of implication relationships.

While context reduction is not always necessary for the algorithm to reach its goal, it reduces the number of negative application conditions (in an equivalent fashion, i.e. without losing information). This in turn reduces the number of potential applications of context generation, speeding up the algorithm.

We formalize context generation as follows:

Lemma 7.33 (context generation)**.** *We describe* context generation *as a function* cg *such that:*

1. *Given two patterns $C = \exists(i_P : \varnothing \hookrightarrow P)$ and $C' = \exists(i_{P'} : \varnothing \hookrightarrow P', ac_{P'})$ with a composed negative application condition $ac_{P'}$ and given an injective morphism $p : P' \hookrightarrow P$, $C^* = \mathrm{cg}(C, C', p)$ is a graph pattern and we have $\neg C' \vDash \neg C^*$.*
2. *Given two patterns $C = \exists(i_P : \varnothing \hookrightarrow P, ac_P)$ and $C' = \exists(i_{P'} : \varnothing \hookrightarrow P', ac_{P'})$ with composed negative application conditions $ac_P = ac'_P \wedge \exists(x : P \hookrightarrow X)$, ac'_P, and $ac_{P'}$ and given an injective morphism $p : P' \hookrightarrow X$, $C^* = \mathrm{cg}(C, C', p)$ is a graph pattern and we have $\neg C \wedge \neg C' \vDash \neg C^*$.*

We write $C \Rightarrow_{\mathrm{cg},C',p} C^*$ to express $C^* = \mathrm{cg}(C, C', p)$. Given a composed graph pattern $\mathcal{C} = \bigwedge_{u \in U} \neg C_u$, we write $C \Rightarrow_{\mathrm{cg},\mathcal{C}} C^*$ to express that there exist $u \in U$ and an appropriate injective morphism p such that $C \Rightarrow_{\mathrm{cg},C_u,p} C^*$.

Construction. *With* $\mathrm{Shift}(p, ac') = \bigwedge_{i \in I} \neg\exists x'_i$, *we construct* cg *as follows:*

$$\mathrm{cg}(C, C', p) = \begin{cases} \exists(i_P, ac_P \wedge \bigwedge_{i \in I} \neg\exists x'_i) & \textit{case 1 (left figure) and no bijective } x'_i \\ \exists(i_P, ac'_P \wedge \bigwedge_{i \in I} \neg\exists(x'_i \circ x)) & \textit{case 2 (right figure) and no bijective } x'_i \\ C & \textit{otherwise, i.e. if there is a bijective } x'_i \end{cases}$$

$$\begin{tikzcd} & P' \triangleleft ac_{P'} \arrow[d, hook, "p"'] & \\ \varnothing \arrow[ur, hook, "i_{P'}"] \arrow[r, hook, "i_P"'] & P \arrow[r, hook, "\triangle ac_P", "x'_i"'] & X'_i \end{tikzcd} \qquad \begin{tikzcd} & & P' \triangleleft ac_{P'} \arrow[d, hook, "p"'] & \\ \varnothing \arrow[urr, hook, "i_{P'}"] \arrow[r, hook, "i_P"'] & P \arrow[r, hook, "\triangle ac'_P", "x"'] & X \arrow[r, hook, "x'_i"'] & X'_i \end{tikzcd}$$

Proof. By Definition 2.36, we need to show that, for all graphs G, $G \vDash \neg C \wedge \neg C'$ implies $G \vDash \neg C^*$, where $\neg C^* = \mathrm{cg}(C, C', p)$. We will focus on the case $\mathrm{cg}(C, C', p) = \exists(i_P, ac'_P \wedge \bigwedge_{i \in I} \neg\exists(x'_i \circ x))$ (see above, right figure) first.

Consider an arbitrary graph G with $G \vDash \neg C \wedge \neg C'$, implying $G \nvDash C$ and $G \nvDash C'$. We need to show $G \nvDash C^*$ and will start from $G \nvDash C$ where we will have to consider two cases:

$$\begin{tikzcd} & & P' \triangleleft ac_{P'} \arrow[d, hook, "p"'] \arrow[dd, hook, bend left, "q'"] & \\ \varnothing \arrow[urr, hook, "i_{P'}"] \arrow[r, hook, "i_P"'] \arrow[drr, hook, bend right, "i_G"'] & P \arrow[r, hook, "\triangle ac'_P", "x"'] \arrow[dr, hook, "q"'] & X \arrow[r, hook, "x'_i"'] \arrow[d, hook, "y"'] & X'_i \arrow[dl, hook, "y'"] \\ & & G & \end{tikzcd}$$

1. There does not exist an injective morphism $q : P \hookrightarrow G$. This implies $G \nvDash C^*$, concluding this case.
2. There exists an injective morphism $q : P \hookrightarrow G$ (and $q \circ i_P = i_G$). By definition of satisfiability and by precondition – $G \nvDash C$ – we have $q \nvDash ac'_P \wedge \neg\exists x$, which gives us $q \nvDash ac'_P \vee q \nvDash \neg\exists x$. We will again distinguish both cases:
   a) We have $q \nvDash ac'_P$. This would lead to $G \nvDash C^*$, concluding this case.

b) We have $q \not\models \neg\exists x$, i.e. $q \models \exists x$, which implies the existence of an injective morphism $y : X \hookrightarrow G$ such that $y \circ x = q$. Then, $q' = y \circ p$ is an injective morphism $q' : P' \hookrightarrow G$ and $q' \circ i_{P'} = i_G$. By precondition – $G \not\models C'$ – we have $q' \not\models ac_{P'}$. By the Shift-lemma, we have $y \not\models \text{Shift}(p, ac_{P'})$ with $\text{Shift}(p, ac_{P'}) = \bigwedge_{i \in I} \neg\exists x'_i$.
Consequently, we have $y \models \exists x_i$ for some i in I. Then, there exists an injective morphism $y' : X'_i \hookrightarrow G$ such that $y' \circ x'_i = y$ (see diagram above) for that specific i. Furthermore, we have $y' \circ x'_i \circ x = y \circ x = q$ and thus, $q \not\models \bigwedge_{i \in I} \neg\exists(x'_i \circ x)$, i.e. $G \not\models C^*$, concluding this case.

Hence, $G \models \neg C \wedge \neg C'$ implies $G \models \neg C^*$ for all graphs G and we have $\neg C \wedge \neg C' \models \neg\,\text{cg}(C, C', p)$. This proof works analogously for the case of morphisms $p : P' \hookrightarrow P$ and $\text{cg}(C, C', p) = \exists(i_P, ac_P \wedge \bigwedge_{i \in I} \neg\exists x'_i)$. If one of the x'_i is bijective, we have $\text{cg}(C, C', p) = C$ and $\neg C \wedge \neg C' \models \neg\,\text{cg}(C, C', p)$ holds trivially. □

When applying the context generation function cg, we need to consider three cases:

As explained above, the first case applies to the first application of cg to the existential condition $\exists i_P$ of a pattern. The graph P contains the positive context graph P' of a pattern C' as a subgraph; then, in order to get to $\neg C' \models \neg C^*$ (equivalent to $C^* \models C'$), we have to extend i_P by the shifted composed negative application condition $\text{Shift}(p, ac_{P'})$.

The second case is applicable to a pattern with a non-trivial composed negative application condition ac_P: we match P' via $p : P' \hookrightarrow X$ to a fitting graph appearing in ac_P. Then, in order to fulfill the required implication, we again shift $ac_{P'}$ via p. Note that P is a subgraph of all potential graphs X appearing in negative application conditions of ac_P; hence, morphisms $p : P' \hookrightarrow P$ are also covered by this case.

The third case prevents generation of redundant context: if a shifted composed negative application contains a bijective morphism in a negative application condition, the result $\text{cg}(C, C', p)$ for the cases above would be equivalent to false and, as such, useless for further context generation steps. To prevent that, cg returns the input pattern as its result. Of course, the implementation has to take care of avoiding subsequent attempts to generate context via morphisms leading to that result – for example, by keeping track of morphisms used by cg.

Example 7.34 (context generation). We consider again Example 7.32 (p. 214) and, in particular, the reduced source pattern $src_{1|\varnothing} = \exists(i_{S_1}, \neg\exists x'_1)$ in Figure 7.20 (p. 216). We start the process of context generation and reduction with $\exists i_{S_1}$. Then, Figure 7.22 shows a morphism $p : P^H_{18} \hookrightarrow S_1$. The corresponding pattern $H_{18} = \exists(i_{P^H_{18}}, \neg\exists x_{18})$ with its non-trivial composed negative application condition $\neg\exists x_{18}$ can be used to generate context. The result

$$C^* = \text{cg}(\exists i_{S_1}, P^H_{18}, p) = \exists(i_{S_1}, \neg x'_1)$$

describes the existence of a controlled shuttle in speed mode slow with a switch two tracks ahead – and the absence of a control unit (c) attached to that switch (t3).

By Lemma 7.33 (p. 218), we have $\neg H_{18} \models \neg C^*$. This makes sense: $\neg H_{18}$, which is equivalent to $\neg\exists(i_{P^H_{18}}, \neg\exists x_{18})$ and $\forall(i_{P^H_{18}}, \exists x_{18})$ requires all switches to have a control unit attached. Then, since S_1 contains a switch, a corresponding control unit must exist: $\forall(i_{S_1}, \exists x'_1)$ must hold, which is equivalent to $\neg C^*$.

Usually, the algorithm would apply one context reduction step (explained below) after generating context. In this example, we skip this step; however, it would not change C^* anyway.

Figure 7.23 describes the second case of context generation. We use the pattern $H_{17} = \exists(i_{P^H_{17}}, \neg\exists x_{17})$ to generate context for the pattern $C' = C^* = \exists(i_{S_1}, \neg x'_1)$ created above. There

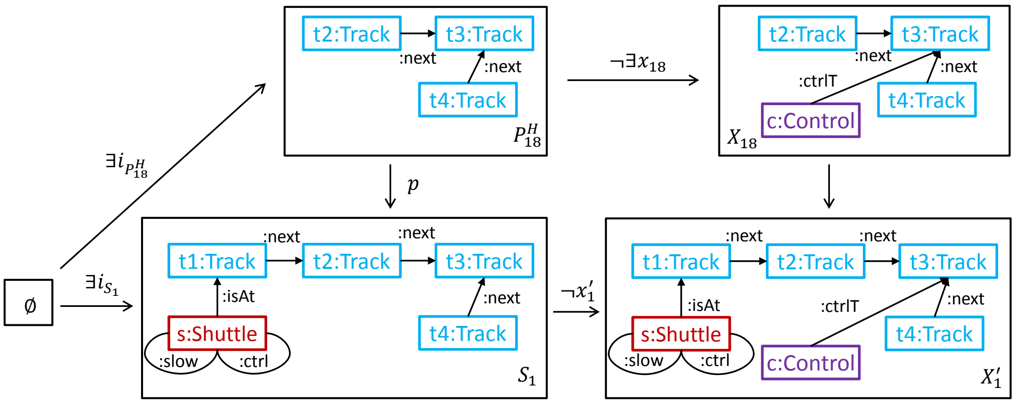


Figure 7.22. – Existential condition $C = \exists i_{S_1}$ of reduced source pattern and result of context generation $\exists(i_{S_1}, \neg\exists x'_1) = \text{cg}(\exists i_{S_1}, P^H_{18}, p : H_{18} \hookrightarrow S_1)$

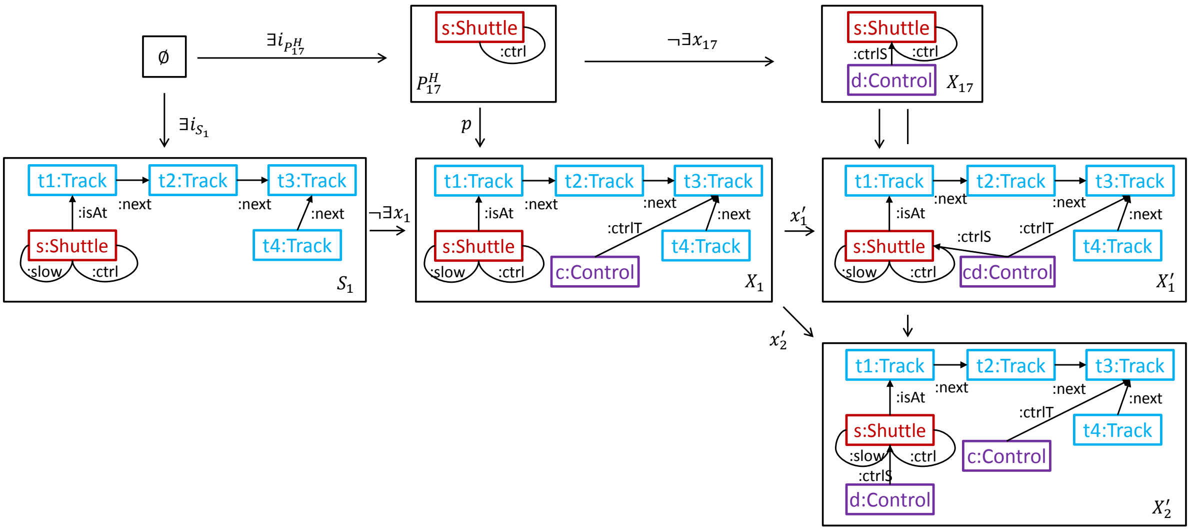


Figure 7.23. – Pattern $C' = \exists(i_{S_1}, \neg\exists x_1)$ and result of context generation $\exists(i_{S_1}, \neg\exists(x'_1 \circ x_1) \wedge \neg\exists(x'_2 \circ x_1)) = \text{cg}(C', H_{17}, p : P^H_{17} \hookrightarrow X_1)$

is an injective morphism $p : P^H_{17} \hookrightarrow X_1$; then

$$C'' = \text{cg}(C', H_{17}, p) = \exists(i_{S_1}, \neg\exists(x'_1 \circ x_1) \wedge \neg\exists(x'_2 \circ x_1))$$

describes the existence of a controlled shuttle in speed mode slow with a switch (t3) two tracks ahead. Its composed negative application condition specifies the absence of a control unit (cd) attached to the shuttle and the switch $(\neg\exists(x'_1 \circ x_1))$ and the absence of two control units (c and d) attached to the shuttle and the switch, respectively $(\neg\exists(x'_2 \circ x_1))$.

By Lemma 7.33 (p. 218), we have $\neg C' \wedge \neg H_{17} \vDash \neg C''$. Again, this was to be expected: $\neg H_{17}$ requires the existence of an attached control unit for every controlled shuttle. The negated pattern $\neg C'$ specifies the absence of the situation in S_1 unless the switch has an attached control unit (as seen in X_1). Since X_1 contains a controlled shuttle, the resulting (negated) pattern $\neg C''$ also needs to take $\neg H_{17}$ into account: it describes the absence of S_1 unless both the shuttle (because of H_{17}) and the track (because of C' and $\neg H_{18}$, see above) have a control unit attached. The two latter conditions can be fulfilled by a single control unit (as in X'_1) and by two separate control units (as in X'_2), hence the two negative application conditions. △

Only applying context generation steps may lead to a growing number of negative application conditions. Context reduction, which is described in the following, offers a way to reduce that number without losing information.

Lemma 7.35 (context reduction). *Let $C = \exists(i_P : \varnothing \hookrightarrow P, ac)$ be a graph pattern with $ac = \bigwedge_{j\in J} \neg\exists(x_j : P \hookrightarrow X_j)$ and let $\mathcal{C} = \bigwedge_{i\in I} \neg\exists(i_{P_i} : \varnothing \hookrightarrow P_i)$ be a composed graph pattern. We describe* context reduction *as a function* cr *such that $C^* = \mathrm{cr}(C,\mathcal{C})$ is a graph pattern and we have $(\mathcal{C} \wedge \neg C) \equiv (\mathcal{C} \wedge \neg C^*)$. We write $C \Rightarrow_{\mathrm{cr},\mathcal{C}} C^*$ to express $C^* = \mathrm{cr}(C,\mathcal{C})$.*

Construction. *With J' the largest subset of J such that for each $j' \in J'$ we have $X_{j'} \vDash \mathcal{C}$, we construct* cr *as follows:*

$$\mathrm{cr}(C,\mathcal{C}) = \exists(i_P, \bigwedge_{j'\in J'} \neg\exists x_{j'})$$

$$\begin{tikzcd}
 & P_i & \\
\varnothing \arrow[ur, hook, "i_{P_i}"] \arrow[r, hook, "i_P"] \arrow[urr, phantom, ""] \arrow[rr, hook, bend right=15, "i_{X_j}"'] \arrow[drr, hook, bend right=30, "i_{X_{j'}}"'] & P \arrow[r, hook, "x_j"] \arrow[r, phantom, shift left=3, "\triangleright ac"] \arrow[dr, hook, "x_{j'}"] & X_j \\
 & & X_{j'}
\end{tikzcd}$$

Proof. By definition of equivalence of graph constraints, we have to show that, for all graphs G, $G \vDash \mathcal{C} \wedge \neg C$ if and only if $G \vDash \mathcal{C} \wedge \neg C^*$ with $\neg C^* = \neg\exists(i_P, \bigwedge_{j'\in J'} \neg\exists x_{j'})$.

$$\begin{tikzcd}
 & P_i \arrow[dr, hook, "y"] & \\
\varnothing \arrow[ur, hook, "i_{P_i}"] \arrow[r, hook, "i_P"] \arrow[dr, hook, "i_G"'] & P \arrow[r, hook, "x_j"] \arrow[r, phantom, shift left=3, "\triangleright ac"] \arrow[d, hook, "q"'] \arrow[dr, hook, "x_{j'}"] & X_j \arrow[dl, hook, "q'"] \\
 & G & X_{j'}
\end{tikzcd}$$

Only if. Consider an arbitrary graph G with $G \vDash \mathcal{C} \wedge \neg C$ and hence, $G \nvDash C$. We have to consider two cases:

1. There does not exist an injective morphism $q : P \hookrightarrow G$. This implies $G \vDash C^*$, concluding this case.
2. There exists an injective morphism $q : P \hookrightarrow G$ (and $q \circ i_P = i_G$). By definition of satisfiability and by precondition – $G \nvDash C$ – we have $q \nvDash ac$. Given $ac = \bigwedge_{j'\in J'} \neg\exists x_{j'} \wedge \bigwedge_{j\in J\setminus J'} \neg\exists x_j$, we have $q \nvDash \bigwedge_{j'\in J'} \neg\exists x_{j'}$ (case 1) or $q \nvDash \bigwedge_{j\in J\setminus J'} \neg\exists x_j$ (case 2).
   We assume $q \nvDash \bigwedge_{j\in J\setminus J'} \neg\exists x_j$ (case 2), implying the existence of an injective morphism $q' : X_j \hookrightarrow G$ such that $q' \circ x_j = q$ for some $j \in J \setminus J'$. Since, by precondition, $X_j \nvDash \mathcal{C}$, there is an i in I such that $X_j \nvDash \neg\exists i_{P_i}$ (for the specific i and j), implying the existence of an injective morphism $y : P_i \hookrightarrow X_j$. Then, $q' \circ y$ is an injective morphism $(q' \circ y) : P_i \hookrightarrow G$ and $q' \circ y \circ i_{P_i} = i_G$, implying $G \nvDash \neg\exists i_{P_i}$ and leading to $G \nvDash \mathcal{C}$, which is a contradiction. Hence, we have $q \vDash \bigwedge_{j'\in J\setminus J'} \neg\exists x_{j'}$, which leads to $q \nvDash \bigwedge_{j'\in J'} \neg\exists x_{j'}$ (case 1), implying $G \nvDash C^*$ and hence, $G \vDash \mathcal{C} \wedge \neg C^*$.

If. Consider an arbitrary graph G with $G \vDash \mathcal{C} \wedge \neg C^*$ and hence, $G \nvDash C^*$. We have to consider two cases:

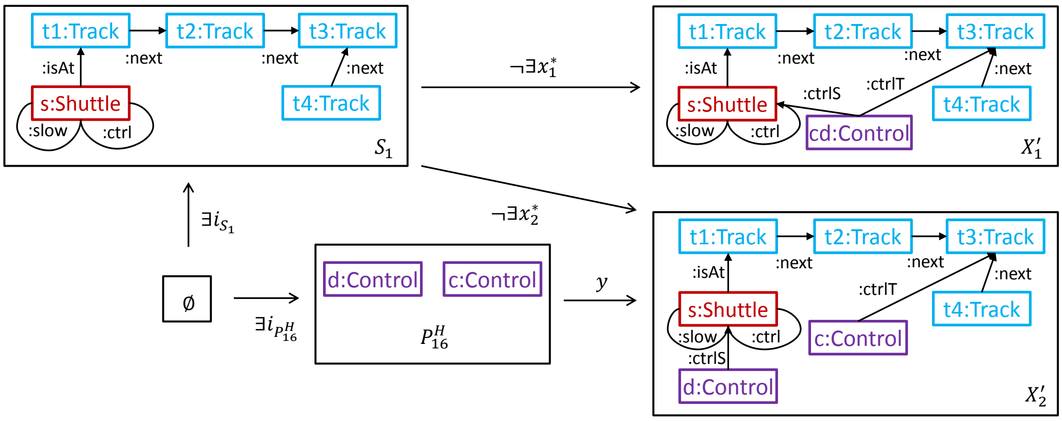


Figure 7.24. – Pattern $C'' = \exists(i_{S_1}, \neg\exists x_1' \wedge \neg\exists x_2')$ and pattern $H_{16} = \exists i_{P_{16}^H}$ with $\exists i_{X_2'} \vDash H_{16}$

1. There does not exist an injective morphism $q : P \hookrightarrow G$. This implies $G \nvDash C$, concluding this case.
2. There exists an injective morphism $q : P \hookrightarrow G$ (and $q \circ i_P = i_G$). By definition of satisfiability and by precondition – $G \nvDash C^*$ – we have $q \nvDash \bigwedge_{j' \in J'} \neg\exists x_{j'}$. Since $ac = \bigwedge_{j' \in J'} \neg\exists x_{j'} \wedge \bigwedge_{j \in J \setminus J'} \neg\exists x_j$, we have $q \nvDash ac$, implying $G \nvDash C$ and $G \vDash \mathcal{C} \wedge \neg C$.

Thus, $G \vDash \mathcal{C} \wedge \neg C$ implies $G \vDash \mathcal{C} \wedge \neg C^*$ for all graphs G and vice versa. In summary, we get $\mathcal{C} \wedge \neg C \equiv \mathcal{C} \wedge \neg C^*$. □

In theory, it is possible for a context reduction step to create a pattern with an empty conjcuntion for its composed negative application condition, which then defaults to the trivial condition true. A subsequent context generation step would have to apply the first case of cg to continue the process. Again, the implementation has to take this into account in order to avoid infinite loops in the algorithm's execution.

Example 7.36 (context reduction)**.** Figure 7.24 shows the application of context reduction to the pattern $C'' = \exists(i_{S_1}, \neg\exists x_1^* \wedge \neg\exists x_2^*)$, which was created in the previous context generation step described in Example 7.34 (p. 219) (with $x_1^* = x_1' \circ x_1$ and $x_2^* = x_2' \circ x_1$). We consider the pattern $H_{16} = \exists i_{P_{16}^H}$, which is part of a composed graph pattern $\mathcal{C}_2 = \neg H_{16} \wedge \neg H_1 \wedge \ldots \wedge \neg H_{15}$ of (negated) forbidden and guaranteed patterns with a trivial composed negative application condition. X_2' contains P_{16}^H, which implies $X_2' \nvDash \neg H_{16}$ and $X_2' \nvDash \mathcal{C}_2$. Since X_1' does not contain P_{16}^H and, indeed, satisfies $\mathcal{C}_2$, we get

$$C^* = \mathrm{cr}(C'', \mathcal{C}_2) = \exists(i_{S_1}, \neg\exists x_1^*).$$

This pattern describes the existence of a controlled shuttle in speed slow with a switch (t3) two tracks ahead – and the absence of a control unit connected to both the shuttle and the switch (t3).

By Lemma 7.35 (p. 221), we have $(\mathcal{C}_2 \wedge \neg C'') \equiv (\mathcal{C}_2 \wedge \neg C^*)$. The negation of C'' requires the existence of a connected control unit for both the shuttle and the switch. Since $\mathcal{C}_2 = \neg H_{16} \wedge \ldots$ forbids the existence of two control units (among other situations irrelevant here), X_1' is the only way to fulfill both $\neg C''$ and $\mathcal{C}_2$; thus, we can discard $\neg x_2^*$ without losing information.

After this step of context reduction, further context generation steps can be applied. However, as we will see in the theorem and example below, C^* is already the result we need to discard the original source pattern as a counterexample. △

From an algorithmic perspective, Lemmas 7.33 (p. 218) and 7.35 (p. 221) describe execution steps without specifying a termination criterion. The idea is to generate (and reduce) context from the existential part of a reduced source or target pattern until the resulting pattern is implied by the original reduced source or target pattern. This requires to check implication of two patterns as described in Theorem 6.8 (p. 120) and Algorithm 6.1 (p. 121). If successful, the chain of implication relationships established by context generation and reduction via Lemmas 7.33 (p. 218) and 7.35 (p. 221) lets us conclude $\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{|\varnothing}$.

Theorem 7.37 (implication of composed graph patterns). *Let $C = \exists(i_P, ac_P)$ be a graph pattern and let $\mathcal{C}_1 = \bigwedge_{i\in I} \neg C_{1,i}$ with $C_{1,i} = \neg\exists(i_{P_{1,i}} : \varnothing \hookrightarrow P_{1,i}, ac_{P_{1,i}})$ and $\mathcal{C}_2 = \bigwedge_{j\in J} \neg C_{2,j}$ with $C_{2,j} = \neg\exists(i_{P_{2,j}} : \varnothing \hookrightarrow P_{2,j})$ be two composed graph patterns.*

$\mathcal{C}_1 \wedge \mathcal{C}_2$ implies $\neg C$ if there is a sequence of alternating applications of context generation (via $\mathcal{C}_1$) and context reduction (via $\mathcal{C}_2$) $\exists i_P \Rightarrow_{\mathrm{cg},\mathcal{C}_1} C' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} \cdots \Rightarrow_{\mathrm{cr},\mathcal{C}_2} C^*$ such that $C \vDash C^*$.

Proof. Consider an arbitrary sequence $\exists i_P \Rightarrow_{\mathrm{cg},\mathcal{C}_1} C' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} \cdots \Rightarrow_{\mathrm{cr},\mathcal{C}_2} C^*$ with $C \vDash C^*$. $C \vDash C^*$ implies $\neg C^* \vDash \neg C$; then, if $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg C^*$, we would get $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg C$. We will show $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg C^*$ by induction.

Base case. Consider $\exists i_P \Rightarrow_{\mathrm{cg},\mathcal{C}_1} C' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} C''$. This implies the existence of an $i \in I$ and an appropriate morphism p such that $C \Rightarrow_{\mathrm{cg},C_{1,i},p} C'$, i.e. $C' = \mathrm{cg}(\exists i_P, C_{1,i}, p)$. By Lemma 7.33 (p. 218) (context generation), we have $\neg C_{1,i} \vDash \neg C'$ and hence, $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg C'$.

Furthermore, $C' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} C''$ implies $\mathcal{C}_2 \wedge \neg C' \vDash \neg C''$, which, together with $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg C'$, implies $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg C''$.

Inductive step. Consider applications of context generation (via $\mathcal{C}_1$) and context reduction (via $\mathcal{C}_2$) $D \Rightarrow_{\mathrm{cg},\mathcal{C}_1} D' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} D''$ with graph patterns D, D', and D'' and, by inductive hypothesis, $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg D$. $D \Rightarrow_{\mathrm{cg},\mathcal{C}_1} D'$ implies the existence of an $i \in I$ and an appropriate morphism p such that $D \Rightarrow_{\mathrm{cg},C_{1,i},p} D'$, i.e. $D' = \mathrm{cg}(D, C_{1,i}, p)$. By Lemma 7.33 (p. 218) (context generation), we have $\neg C_{1,i} \wedge \neg D \vDash \neg D'$. With $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg D$, we get $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg D'$.

Furthermore, $D' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} D''$ implies $\mathcal{C}_2 \wedge \neg D' \vDash \neg D''$, which, together with $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg D'$, implies $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg D''$, concluding the inductive proof. □

Note that, as mentioned earlier in this section, an implementation of this approach will not necessarily terminate. This problem is partly a result of the undecidability of the underlying general problem; however, it is also a question of the order of execution when generating context. If context generation can be applied for multiple morphisms and constraints, some parameters may yield a solution while others allow an arbitrarily large number of subsequent generation steps, effectively preventing termination. Still, as discussed in this section's examples and in the evaluation (Chapter 9), using context generation can yield positive results (in finite time) where our previous approach to implication results in false negatives.

Example 7.38 (implication of composed graph patterns). Consider $C^* = \exists(i_{S_1}, \neg\exists x_1^*)$ in Example 7.36, Figure 7.24 (p. 222). We have created C^* by applying context generation and context reduction, starting from $\exists i_{S_1}$, which was part of a reduced source pattern $src_{1|\varnothing} = \exists(i_{S_1}, \neg\exists x_{S_1})$ (Figure 7.20). Given separation of $\mathcal{F} \wedge \mathcal{H}$ in composed graph patterns $\mathcal{C}_1$ and $\mathcal{C}_2$ as required by Theorem 7.37 (p. 223), we have (from Examples 7.34 (p. 219) and 7.36 (p. 222)):

$$\exists i_{S_1} \Rightarrow_{\mathrm{cg},\mathcal{C}_1} C' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} C' \Rightarrow_{\mathrm{cg},\mathcal{C}_1} C'' \Rightarrow_{\mathrm{cr},\mathcal{C}_2} C^*.$$

By Theorem 6.8 (p. 120), $src_{1|\varnothing}$ implies $\vDash C^*$; in fact, all their components are isomorphic. By Theorem 7.37 (p. 223), we have $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg src_{1|\varnothing}$ and, with $\mathcal{C}_1 \wedge \mathcal{C}_2 = \mathcal{F} \wedge \mathcal{H}$, we get $\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{1|\varnothing}$. △

In order to establish a connection between checking implication involving composed graph patterns and discarding counterexamples, we refine Theorem T.2r (p. 143). The difference lies in the conditions sufficient to discard a counterexample: instead of showing $src_{|\varnothing} \vDash F_i$ or $src_{|\varnothing} \vDash H_j$ for a source pattern in an s/t-pattern sequences and $i \in I$ and $j \in J$, we attempt to prove $\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{|\varnothing}$ for any source pattern. The analysis of the rightmost target pattern $tar_{k|\varnothing}$ has been changed similarly: instead of $tar_{k|\varnothing} \vDash H_j$, the approach checks $\mathcal{H} \vDash \neg tar_{k|\varnothing}$.

Theorem T.2e-ai (k-inductive invariant checking and implication of composed patterns)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ and $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$ be a composed forbidden pattern and composed guaranteed pattern, respectively.*

$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ if, for all sequences $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ with $seq \in \mathrm{Seq}_k^r(\mathcal{R}, \neg\mathcal{F})$, one of the following conditions holds:

1. $\exists z(1 \le z \le k \wedge (\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{z|\varnothing}))$.
2. $\mathcal{H} \vDash \neg tar_{k|\varnothing}$.

Proof. According to Lemma 6.10, we need to show that for all k-sequences of transformations $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$, it holds that:

$$\exists u(G_k \vDash F_u) \Rightarrow \exists z, v(0 \le z \le k \wedge G_z \vDash H_v) \vee \exists z, v(0 \le z \le k-1 \wedge G_z \vDash F_v)$$

Consider an arbitrary k-sequence of transformations to $\mathcal{R}$ (with corresponding graphs) $trans = G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ such that $\exists u(G_k \vDash F_u)$ with, for ease of reading, $F_u = F$. More specifically, $trans = G_0 \Rightarrow_{b_1,m_1,m_1'} \dots \Rightarrow_{b_k,m_k,m_k'} G_k$ for rules $b_i \in \mathcal{R}$ and matches (comatches) m_i (m_i') and $trans$ leads to F. We want to show that $G_z \vDash H_v$ for $0 \le z \le k$ and $v \in J$ or that $G_z \vDash F_v$ for $0 \le z \le k-1$ and $v \in I$.

By Theorem T.1r, there is a k-sequence of s/t-patterns $seq \in \mathrm{Seq}_k(\mathcal{R}, F)$ with $trans \vDash seq$. Then, $seq' = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_k} tar_k$ and $trans \vDash seq'$ (Lemma 7.3). By precondition, one of the following is true:

1. There exists a z with $1 \le z \le k$ such that $\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{z|\varnothing}$. Because of $trans \vDash seq$, we have $m_z \vDash src_z$ and, with $m_z : L_z \hookrightarrow G_{z-1}$ and Lemma 2.38, we gain $G_{z-1} \vDash src_{z|\varnothing}$. By contraposition and by implication of graph constraints (Definition 2.36 (p. 42)) we get $G_{z-1} \nvDash \mathcal{F} \wedge \mathcal{H}$, implying the existence of a v such that $G_{z-1} \vDash F_v$ or $G_{z-1} \vDash H_v$.
2. We have $\mathcal{H} \vDash tar_{k|\varnothing}$. Because of $trans \vDash seq$, we have $m_k' \vDash tar_k$ and, with $m_k' : R_k \hookrightarrow G_k$ and Lemma 2.38, we gain $G_k \vDash tar_{k|\varnothing}$. By contraposition and by implication of graph constraints (Definition 2.36 (p. 42)), we get $G_k \nvDash \mathcal{H}$, implying the existence of a v such that $G_k \vDash H_v$.

Hence, $\mathcal{F}$ is a k-inductive invariant for GTS under H. □

From an algorithmic perspective, it is advisable to perform the implication check described by Theorem 6.8 (p. 120) and Algorithm 6.1 (p. 121) before using context generation (and context reduction) to show implication of a negated reduced source or target pattern by a composed graph pattern. It is much more efficient to discard a reduced source pattern by implication of an individual forbidden or guaranteed pattern. In the worst case, the algorithm to check implication by context generation might not terminate (unless forced to) while checking implication by individual patterns would yield a positive result (in finite time, because Algorithm 6.1 (p. 121) always terminates).

Example 7.39 (2-inductive invariant checking and implication of composed patterns for a safe system)**.** In Example 7.38 (p. 223), we have shown $\mathcal{F} \wedge \mathcal{H} \vDash \neg src_{1|\varnothing}$. By Theorem T.2e-ai, this disqualifies the corresponding sequence (cf. Example 7.32 (p. 214)) as a potential counterexample. All s/t-pattern sequences in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$ that could not be discarded using implication of patterns (Theorem 6.8 (p. 120) and Algorithm 6.1 (p. 121)) can be discarded by showing implication of a (negated) reduced source pattern by $\mathcal{F} \wedge \mathcal{H}$ via context generation.△

We can also refine Theorem T.3r (p. 149) (k−1-bounded backward model checking) to take implication of composed patterns into account. Again, only the three conditions to discard s/t-pattern sequences are modified. Theorems T.4r (p. 153) and T.5r (p. 155) can then be adjusted in the same fashion.

Theorem T.3e-ai (k−1-bounded backward model checking and implication of composed patterns)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$, $\mathcal{H} = \bigwedge_{j\in J} \neg H_j$, and $\mathcal{S} = \bigwedge_{o\in O} \neg S_o$ be a composed forbidden pattern, composed guaranteed pattern, and composed start configuration pattern, respectively, with $\mathcal{S} \vDash \mathcal{F}$.*

For all graphs $G \in \mathrm{REACH}_{k-1}(GG, \mathcal{H})$ and graph grammars $GG = (GTS, G_0)$ with $GG \in \mathrm{IND}(GTS, \mathcal{S})$, we have $G \vDash \mathcal{F}$, if for all sequences $seq = src_1 \Rightarrow_{b_1} \dots \Rightarrow_{b_n} tar_n$ with $seq \in \mathrm{SEQ}^r_{k-1}(\mathcal{R}, \neg\mathcal{F})$, one of the following conditions holds:

1. $\exists z(1 \leq z \leq n \wedge (\mathcal{H} \vDash \neg src_{z|\varnothing}))$.
2. $\mathcal{H} \vDash \neg tar_{k|\varnothing}$.
3. $\mathcal{S} \wedge \mathcal{H} \vDash \neg src_{1|\varnothing}$.

Proof. This follows by adjusting the proof to Theorem T.3r (p. 149) in a fashion similar to the above adjustments to the proof of Theorem T.2r (p. 143). □

Implication of composed patterns has another possible application beneficial to performance and possibly, to completeness. If a composed forbidden graph pattern $\mathcal{F}$ can be split into $\mathcal{F}_1 \wedge \mathcal{F}_2$ such that $\mathcal{F}_1 \wedge \mathcal{H}$ implies all individual negated patterns in $\mathcal{F}_2$, we only need to verify $\mathcal{F}_1$ as a k-inductive invariant under $\mathcal{H}$ to conclude that $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$:

Theorem 7.40 (eliminating forbidden patterns)**.** *Given a composed forbidden pattern $\mathcal{F}$ such that $\mathcal{F} = \mathcal{F}_1 \wedge \mathcal{F}_2$ with $\mathcal{F}_2 = \bigwedge_{i\in I} \neg F_i$, a composed guaranteed pattern $\mathcal{H}$, and a graph transformation system GTS. $\mathcal{F}$ is a k-inductive invariant of GTS under $\mathcal{H}$ if the following conditions are fulfilled:*

1. *$\mathcal{F}_1$ is a k-inductive invariant of GTS under $\mathcal{H}$.*
2. $\forall i((i \in I) \Rightarrow (\mathcal{F}_1 \wedge \mathcal{H} \vDash \neg F_i))$.

Proof. Consider an arbitrary transformation sequence to $\mathcal{R}$ $trans = G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_k$ with $G_z \vDash \mathcal{F}$ for $1 \leq z \leq k-1$ and $G_z \vDash \mathcal{H}$ for $1 \leq z \leq k$. By Definition D.1 (p. 62), we have to show $G_k \vDash \mathcal{F}$.

Since $\mathcal{F}_1$ is a k-inductive invariant under $\mathcal{H}$ (1), we have $G_k \vDash \mathcal{F}_1$. Given $G_k \vDash \mathcal{F}_1$, $G_k \vDash \mathcal{H}$, and $\forall i((i \in I) \Rightarrow (\mathcal{F}_1 \wedge \mathcal{H} \neg \vDash F_i))$ (2), we get $G_k \vDash \neg F_i$ for all $i \in I$. Thus, we have $G_k \vDash \mathcal{F}_2$, implying $G_k \vDash \mathcal{F}_1 \wedge \mathcal{F}_2$ and hence, $\mathcal{F} = \mathcal{F}_1 \wedge \mathcal{F}_2$ is a k-inductive invariant of GTS under $\mathcal{H}$. □

If large graph patterns – or patterns with large negative application conditions – are implied by smaller forbidden and guaranteed patterns, this variant can siginificantly improve performance. The same principle can be applied to k−1-bounded backward model checking. In both cases, an implementation has to find an appropriate way to split of $\mathcal{F}$, which is not trivial: if the algorithm ends up with a set $\mathcal{F}_2$ that contains fewer patterns than could be implied by $\mathcal{F}_1 \wedge \mathcal{H}$, the positive effect is reduced; if the algorithm attempts to show implication for too many patterns, the failed attempts' negative impact on performance may exceed the benefit.

An analysis of possible heuristics is beyond the scope of this thesis; when applying this technique in existing work [6], implication was attempted for patterns whose composed negative application conditions exceeded a fixed size of nodes.

A similar technique can be applied to reduce the computational effort of $k-1$-bounded backward model checking. Given $\mathcal{S}$, $\mathcal{H}$, and $\mathcal{F} = \mathcal{F}_1 \wedge \mathcal{F}_2$ with $\mathcal{F}_2 = \bigwedge_{i \in I} \neg F_i$ and $\mathcal{S} \vDash \mathcal{H}$ such that $\mathcal{F}_1$ holds in $\text{REACH}_{k-1}(GG, \mathcal{H})$ for all $GG \in \text{IND}(GTS, \mathcal{S})$, then $\mathcal{F}$ holds in those state spaces too, if $\forall i((i \in I) \Rightarrow (\mathcal{F}_1 \wedge \mathcal{H} \vDash \neg F_i))$.

Finally, implication with composed patterns can be used to show $\mathcal{S} \vDash \mathcal{F}$ by showing $\mathcal{S} \wedge \mathcal{H} \vDash F_i$ for all pattern F_i in $\mathcal{F}$. This is required for properly establishing the base case of our inductive argument. The same applies for verification of $\mathcal{S} \vDash \mathcal{H}$, if required.

7.4.2. Implementation

In order to implement k-inductive invariant checking and $k-1$-bounded backward model checking with respect to implication for composed graph patterns, we can reuse most of the algorithms for the restricted approach (Section 6.7). We only need to refine discardPattern (Algorithm 6.6 (p. 158)), which was used to discard source or target patterns and, as a result, their s/t-pattern sequences. Here, we extend this function by the implication check described in Theorem 7.37 (p. 223). This is shown in Algorithms 7.9. The implication check for individual patterns (Theorem 6.8 (p. 120), Algorithm 6.1 (p. 121)) is still executed: if source or target patterns can be discarded by comparing them to individual patterns, we do not need to use the implication check with context generation and reduction described in Algorithm 7.10.

Algorithm 7.9: discardPattern(C,$\mathcal{C}$)

```
description: checks implication for graph patterns and for composed graph patterns
input      : a graph pattern $C$, a set $\mathcal{C}$ of forbidden/guaranteed patterns
output     : whether $C$ implies any pattern in $\mathcal{F}$ or $\mathcal{H}$ or $F \wedge \mathcal{H}$ imply $\neg C$
foreach $C' \in \mathcal{C}$ do
    if implies($C$,$C'$) then   /* Theorem 6.8 (p. 120), Algorithm 6.1 (p. 121) */
        return true
return implies($C$,$\mathcal{C}$)                          /* Algorithm 7.10 */
```

Algorithm 7.10 first (lines 1–7) splits the composed graph pattern $\mathcal{C}$ (equal to $\mathcal{F} \wedge \mathcal{H}$ for k-inductive invariant checking) into two sets of patterns $\mathcal{C}_1$ and $\mathcal{C}_2$. As required by Theorem 7.37 (p. 223), $\mathcal{C}_1$ contains only patterns with a non-trivial composed negative application condition while $\mathcal{C}_2$ contains patterns with the trivial composed negative application condition true. The former are used for context generation (line 18), the latter for context reduction (line 23). Note that the algorithm keeps track of morphisms used to generate context in order to not generate redundant context, which could possibly lead to infinite loops. As soon as the algorithm fails to generate context with any combination of patterns and morphisms, it reports failure to conclude $\mathcal{C}_1 \wedge \mathcal{C}_2 \vDash \neg C_{pg}$. This algorithm can also be applied in the fashion described in Theorem 7.40 (p. 225), not only as part of the analysis of s/t-pattern sequences.

The possibility of non-termination lies in the iterative generation of context: an application of cg could generate new context, which in turn triggers generation of new context in the next iteration, and so on. In practice, this is prevented by the loop condition in line 12: if the number of used morphisms (and hence, generation steps) reaches a certain threshold, the algorithm terminates with a return value of false. Runtime or the size of nodes in negative application conditions could be used as thresholds as well.

Algorithm 7.10: implies(C_{pg}, $\mathcal{C}$)

```
description: implication check with composed graph patterns in the sense of
             Theorem 7.37 (p. 223).
input      : a graph pattern $C_{pg} = \exists(i_P : \varnothing \hookrightarrow P, ac_{pg})$, a set of graph patterns $\mathcal{C}$,
             and a configuration parameter threshold to enforce termination
output     : whether or not $\mathcal{C} \models \neg C_{pg}$ can be shown

$\mathcal{C}_1 \leftarrow \varnothing$
$\mathcal{C}_2 \leftarrow \varnothing$
foreach pattern $C'$ in $\mathcal{C}$ do          /* separating patterns by their structure */
    if $C'$ has the form $\exists i_{P'}$ then                  /* trivial nested ac true */
        $\mathcal{C}_2 \leftarrow \mathcal{C}_2 \cup \{C'\}$
    else                                            /* non-trivial nested ac */
        $\mathcal{C}_1 \leftarrow \mathcal{C}_1 \cup \{C'\}$
$ac_P \leftarrow \neg\exists(id_P : P \hookrightarrow P)$        /* dummy condition for first iteration only */
$C \leftarrow \exists i_P$
usedMorphisms $\leftarrow \varnothing$
generated $\leftarrow$ true
while generated and $|usedMorphisms| < threshold$ do
    generated $\leftarrow$ false
    foreach $C' \in \mathcal{C}_1$ with $C' = \exists(i_{P'} : \varnothing \hookrightarrow P', ac_{P'})$ and not generated do
        foreach condition $\neg\exists(x : P \hookrightarrow X)$ in $ac_P$ and not generated do
            foreach morphism $p : P' \hookrightarrow X$
              with $p \notin usedMorphisms$ and not generated do      /* or $p : P' \hookrightarrow P$ */
                usedMorphisms $\leftarrow$ usedMorphisms $\cup \{p\}$
                tmpPattern $\leftarrow$ cg($C, C', p$)                          /* Lemma 7.33 */
                if tmpPattern $= C$ then
                    continue
                else
                    generated $\leftarrow$ true
                    $C \leftarrow$ cr(tmpPattern, $\mathcal{C}_2$)                  /* Lemma 7.35 */
                    if implies($C_{pg}, C$) then
                        return true
return false       /* if an iteration failed to generate further context */
```

7.5. Combining Extensions

The nature of the implementation partially described in Sections 6.7, 7.1.3, 7.2.3, 7.3.4, and 7.4.2 also allows the combination of the restricted approach with multiple extensions. In particular, extensions can be freely combined – with one caveat: in the current implementation, forward propagation (Section 7.1) and implication with composed graph patterns (Section 7.4) do not support patterns with partial negative application conditions (Section 7.3). Hence, composed partial negative application conditions have to be expanded (to composed total negative application conditions) before forward propagation can be applied. Since partial negative application conditions are expanded as part of the Seq-construction anyway (Theorem 7.28 (p. 199)), this may not make a huge difference in most cases, but some of the gain in performance may be lost. Similarly, partial negative application conditions have to be expanded before implication with composed graph patterns via context generation can be applied.

Theorem T.2e combines the restricted approach to k-inductive invariant checking and all the extensions discussed in this chapter. The differences to Theorem T.2r (p. 143) include

- the use of partial negative application conditions ($\mathrm{Seq}_k^{r,p}(\mathcal{R}, \neg\mathcal{F})$) and the subsequent expansion to composed total application conditions (expand and $\mathcal{SQ}(\mathcal{R}, \neg\mathcal{F})$),
- the application of forward propagation ($\mathrm{prop}(\mathit{seq})$),
- the inclusion of conjunctions of negated rule applicability constraints for rules of higher priority ($\mathcal{A}(b_z)$), and
- the analysis of reduced source and target patterns for implication by composed graph patterns ($\mathcal{F} \wedge \mathcal{H} \wedge \mathcal{A}(b_z) \vDash \neg \mathit{src}_{z|\varnothing}$) instead of (individual) graph patterns.

Theorem T.2e (k-inductive invariant checking with extensions)**.** *Let* $\mathit{GTS} = (\mathit{TG}, (\mathcal{R}, \mathit{prio}))$ *be a graph transformation system with priorities and* $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ *and* $\mathcal{H} = \bigwedge_{j \in J} \neg H_j$ *be a composed forbidden pattern and composed guaranteed pattern, respectively.*

Let $\mathcal{A}(b)$ *be the conjunction of negated rule applicability constraints for rules of higher priority:* $\mathcal{A}(b) = \bigwedge_{u \in U} \neg\exists(i_{L_u}, \mathit{ac}_{L_u} \wedge \mathrm{Appl}(b_u))$ *with* $U = \{u \mid b_u \in \mathcal{R} \wedge \mathit{prio}(b_u) > \mathit{prio}(b)\}$.

Given a source or target pattern $\mathit{pat} = \exists(c, \mathrm{PShift}(c^+, \mathit{ac}_C) \wedge \mathit{ac})$ *with a composed partial negative application condition* $\mathrm{PShift}(c^+, \mathit{ac}_C)$ *and a composed (total) negative application condition* ac*, we define* $\mathrm{expand}(\mathit{pat}) = \exists(c, \mathrm{Shift}(c^+, \mathit{ac}_C) \wedge \mathit{ac})$.

Furthermore, we define $\mathrm{Seq}_k^{r,ep}(\mathcal{R}, \neg\mathcal{F}) = \{\mathrm{expand}(\mathit{src}'_1) \Rightarrow_{b_1} (\mathrm{expand}(\mathit{tar}'_1), \mathrm{expand}(\mathit{src}'_2)) \Rightarrow_{b_2} .. \Rightarrow_{b_k} \mathit{tar}_k \mid (\mathit{src}'_1 \Rightarrow_{b_1} (\mathit{tar}'_1, \mathit{src}'_2) \Rightarrow_{b_2} \ldots \Rightarrow_{b_k} \mathit{tar}_k) \in \mathrm{Seq}_k^{r,p}(\mathcal{R}, \neg\mathcal{F})\}$.

$\mathcal{F}$ *is a* k*-inductive invariant for* GTS *under* $\mathcal{H}$ *if, for all sequences* $\mathrm{prop}(\mathit{seq})$ *with* $\mathrm{prop}(\mathit{seq}) = \mathit{src}_1 \Rightarrow_{b_1} \ldots \Rightarrow_{b_k} \mathit{tar}_k$ *and* $\mathit{seq} \in \mathrm{Seq}_k^{r,ep}(\mathcal{R}, \neg\mathcal{F})$*, one of the following conditions holds:*

1. $\exists z(1 \leq z \leq k \wedge (\mathcal{F} \wedge \mathcal{H} \wedge \mathcal{A}(b_z) \vDash \neg \mathit{src}_{z|\varnothing}))$.
2. $\mathcal{H} \vDash \neg \mathit{tar}_{k|\varnothing}$.

Proof. This follows from the proofs of various incarnations of Theorem T.2r (p. 143) (restricted approach) – Theorem T.2e-fp (forward propagation), Theorem T.2e-rp (rules with priorities), and Theorem T.2e-ai (implication with composed graph patterns). We also need Theorem 7.28 (construction of s/t-pattern sequences and equivalence of $\mathrm{Seq}_k^{r}(\mathcal{R}, \neg\mathcal{F})$ and $\mathrm{Seq}_k^{r,p}(\mathcal{R}, \neg\mathcal{F})$) and Lemma 7.20 (equivalence of $\mathrm{PShift}(c, \mathit{ac}_C)$ and $\mathrm{Shift}(c, \mathit{ac}_C)$). □

Note that the effect of partial negative application conditions on performance is not discernible in this theorem. Instead, the benefit comes from the potential of discarding patterns and s/t-pattern sequences by intermediate analysis during the construction of sequences. This has been discussed as part of the implementation of k-inductive invariant checking for the restricted approach (Section 6.7) and with partial negative application conditions (Section 7.3.4).

Theorem T.3r (p. 149) (k−1-bounded backward model checking) could be combined with these extensions in a similar fashion.

Table 7.1. – Properties of the general and restricted approaches and effects of extensions

	Approach/extension	**Appl.-soundness**	**Appl.-termination**	**Appl.-deg.-completeness**	**Appl.-performance**
	General approach	✓	(✓)	?/×	?
	Restricted approach ...	✓	✓	?/×	?
... with	forward propagation	✓	✓	+/×	+ and -
	rule priorities	✓	✓	n.a./×	(n.a.)
	partial negative application conditions	✓	✓	n.a./×	+
	implication with composed patterns	✓	(✓)	+/×	+ and -

7.6. Discussion and Conclusion

Each of the four extensions introduced above serves a specific purpose or addresses a drawback of the restricted approach. In this section, we will shortly discuss the role of the extensions and their effects.

In comparison to the restricted approach, the formal model used in the extensions has only one additional aspect: if rule priorities (Section 7.2) are used, system behavior is specified by graph rules with priorities (and left composed negative application conditions). Other than that, the extensions do not modify the formal model.

Table 7.1 shows the impact and expected effect of the extensions in comparison to the restricted approach with respect to the properties **Appl.-soundness**, **Appl.-termination**, **Appl.-deg.completeness**, and **Appl.-performance**. Again, this is a prognosis based on this chapter's results, with an evaluation to follow in Chapter 9.

With the exception of implication with composed patterns, all extensions have been shown to be sound and terminate by construction and implementation. For implication with composed patterns, the same argument used for the general approach applies: by introducing a threshold on runtime or context generation steps, termination can be enforced – and if inconclusive results are treated as counterexamples, the approach remains sound.

Forward propagation was introduced specifically to reduce the number of false negatives caused by insufficient information in patterns (cf. Section 6.8). As such, the intention and expectation is an improvement of the degree of completeness. The same holds for implication with composed patterns: this extension is meant to address false negatives caused by interactions of forbidden and guaranteed patterns not considered by the implication check of Theorem 6.8 (p. 120). However, both extensions will still not result in completeness in the general case given the underlying undecidable problem. Using rule priorities and partial negative application conditions will not affect completeness in comparison to the restricted approach. The former extension is meant to allow more natural specification of systems with prioritized behavior; the latter extension should generally improve performance of the approach and reduce computational effort.

Given the non-binary nature of **Appl.-performance**, the extensions' effects are more difficult to predict. As mentioned before, partial negative application conditions should have a positive effect on performance. Rule priorities only affect performance in the sense that simulation of priorities by application conditions would lead to more expensive computations during execution of the Seq-construction. Forward propagation, on the other hand, can have both positive and negative effects on performance: by improving the degree of completeness, false counterexamples are discarded and do not have to be considered in further iterations of the Seq-construction for higher values of k. However, forward propagation requires iteration over all source and target patterns of all s/t-pattern sequences – sometimes without the desired result (of discarding a sequence). Similarly, implication with composed patterns can eliminate false negatives, which improves performance, while its execution can be costly. Implication with composed patterns can have a particularly strong impact on performance if forbidden patterns are discarded by other forbidden and guaranteed patterns (Theorem 7.40 (p. 225)) before executing the Seq-construction. For each forbidden pattern F (from the composed forbidden pattern $\mathcal{F}$) discarded in this fashion, we do not have to compute $\mathrm{Seq}_k^r(\mathcal{R}, F)$. The larger the forbidden pattern in question, the higher the benefit.

This concludes the formalization and implementation of several extensions to the restricted approach. More detailed results with respect to the degree of completeness and performance will be discussed in Chapter 9.

8. Related Work

There is a number of other approaches focused on formal verification of graph transformation systems, supporting varying formalisms and varying levels of automation and implementation. In general, explicit-state model checking such as in GROOVE [GdMR$^+$12], Henshin [ABJ$^+$10], or CheckVML [SV03] can only be applied for finite state spaces and a single start graph. Iterative application of explicit-state model checking may be an option for a finite set (of reasonable size) of start graphs. However, they also bring advantages: computing explicit state spaces allows verification of temporal properties instead of properties relating to states – i.e. graphs – only. Also, with all the information of the (finite) state space available, false negatives or false positives in the sense discussed here cannot occur.

If state spaces are represented and analyzed by abstraction, approaches may be applicable to systems with infinite state spaces, but often come with limitations with respect to specifying behavior and properties. The list of such approaches includes the tool Augur (and its second version) [KK08, BCK08]; it analyzes graph transformation systems by representing them as so-called Petri graphs. While the approach can address infinite state spaces, several restrictions with respect to specifications apply: rules may not delete nodes, but must at least delete one edge; application conditions are not supported. Analysis is limited to one initial graph at a time.

An approach by Stückrath and König [KS12, KS14, Stü16, KS17], which is implemented in the tool UNCOVER, also addresses infinite-state systems. By establishing that the graph transformation system in question is well-structured, the approach draws conclusions about the state space. The implementation supports sets of error graphs and initial graphs and backward analysis to reason about the existence of paths to error graphs. However, the approach faces restrictions with respect to transformation systems and, in particular, to negative application conditions, which interfere with the requirements on well-structuredness – or lead to an over-approximation during verification.

Steenken, Wehrheim, and Wonisch have proposed to verify infinite-state graph grammars by abstraction [Ste15, SWW11, SW11]; we will discuss this approach in more detail in Section 8.3.

Symbolic approaches, which do not explicitly or implicitly compute and analyze state spaces, are usually capable of addressing infinite state spaces and potentially infinite sets of start graphs, but may struggle with undecidability, the resulting risk of non-termination of an automated procedure, and perfomance. The list of symbolic approaches includes the approach and implementation by Pennemann and Habel [Pen09, Pen08a, Pen08b, HP09], which addresses verification of graph programs with postconditions and preconditions; it will be discussed in Section 8.2.

A closely related approach by Poskitt and Plump [PP10, PP12, PP13, PP14, Pos13] focuses on Hoare-style verification of graph programs with respect to postconditions and preconditions. It supports more expressive properties than the approach by Pennemann and Habel; however, it is less focused on fully automated execution. Instead, the approach suggests that difficult parts of a proof could be solved with the help of human interaction. The idea is to have parts of the approach implemented in an interactive proof assistant [Pos13] such as Isabelle [NPW02].

Outline. This chapter is structured as follows:

In Section 8.1, we will discuss the relation between 1-induction and k-induction and whether the latter can be performed by the former. The question is whether tools capable of performing

1-inductive invariant checking, most notably Enforce [Pen09] and the earlier incarnation and implementation [BBG⁺06, Dyc12] of the approach proposed by this thesis can also be used to execute k-inductive invariant checking. As a representative for fully symbolic techniques, said approach by Pennemann and Habel [Pen09] will be discussed in Section 8.2; we will also elaborate how it may or may not be applied to perform k-induction. As an example for verification by abstraction, Section 8.3 will discuss the approach by Steenken, Wehrheim, and Wonisch [Ste15].

8.1. 1-Induction and k-Induction

This thesis's main focus is the extension of an approach for 1-inductive invariant checking [BBG⁺06, Dyc12], most notably to k-induction. As discussd in Chapters 5 and 6, this extension via the general and restricted approach is based on constructing s/t-pattern sequences, which are symbolic representations of transformation sequences. However, even without a specific approach, we can attempt to establish a relationship between 1-induction and k-induction. This is relevant because it may suggest that verification approaches capable of performing 1-inductive invariant checking may be employed to perform (a limited form of) k-induction under certain circumstances.

First, any n_1-inductive invariant for a specific value of n_1 is a n_2-inductive invariant for any (integer) value of n_2 larger than n_1:

Lemma 8.1. *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and two graph constraints $\mathcal{F}$ and $\mathcal{H}$. If $\mathcal{F}$ is a n_1-inductive invariant of GTS under $\mathcal{H}$, then, for any n_2 with $n_2 > n_1$, $\mathcal{F}$ is a n_2-inductive invariant for GTS under $\mathcal{H}$.*

Proof. By Definition D.1, for all transformation sequences $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_{n_1}$ we have:

$$(\forall z (0 \le z \le n_1 \Rightarrow G_z \vDash \mathcal{H}) \wedge \forall z (0 \le z \le n_1 - 1 \Rightarrow G_z \vDash \mathcal{F})) \quad \Rightarrow \quad (G_{n_1} \vDash \mathcal{F}).$$

Now, we consider all transformation sequences $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_{n_2}$ with $G_i \vDash \mathcal{H}$ for $0 \le i \le n_2$ and $G_i \vDash \mathcal{F}$ for $0 \le i \le n_2 - 1$. We need to show $G_{n_2} \vDash \mathcal{F}$. Since $n_2 > n_1$, we can split all such sequences into $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_{n_1} \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_{n_2}$. By the statement above we can conclude $G_{n_1} \vDash \mathcal{F}$ and by inductive argument, we can extend this conclusion to $G_i \vDash \mathcal{F}$ for $n_1 + 1 \le i \le n_2$, concluding the proof. □

More specifically, this also applies to 1-induction: any 1-inductive invariant is a k-inductive invariant for any value of k (with $k \ge 1$). A similar (if rather obvious) result can be obtained for k−1-bounded backward model checking: given $n_2 > n_1$, if $G \vDash \mathcal{F}$ for all graphs $G \in \text{REACH}_{n_2}(GG, \mathcal{H})$ and for any graph grammar GG and constraints $\mathcal{F}$ and $\mathcal{H}$, then $G \vDash \mathcal{F}$ for all graphs $G \in \text{REACH}_{n_1}(GG, \mathcal{H})$ – since $\text{REACH}_{n_1}(GG, \mathcal{H}) \subseteq \text{REACH}_{n_2}(GG, \mathcal{H})$.

However, the other direction is much more interesting: we wonder whether iterative execution of 1-induction may allow conclusions regarding a k-inductive invariant. The general idea is to use counterexamples for a 1-inductive invariant and continue from there. Consider the violation of a 1-inductive invariant $\mathcal{F}_1$ under $\mathcal{H}$: a graph transformation $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$, $G' \vDash \mathcal{H}$, $G \vDash \mathcal{F}_1$, and $G' \nvDash \mathcal{F}_1$. Note that this is the level of specific graphs and graph transformations, not symbolic counterexamples (i.e. s/t-pattern sequences).

One idea to address these counterexamples is to include them in the targeted 1-inductive invariant. If we can find a graph constraint $\mathcal{F}_2$ such that all graphs G satisfying the above criteria do not satisfy $\mathcal{F}_2$, we can attempt to show that $\mathcal{F}_1 \wedge \mathcal{F}_2$ is a 1-inductive invariant. All previous counterexamples $G \Rightarrow_{\mathcal{R}} G'$ will be discarded (since $G \nvDash \mathcal{F}_2$). However, we may have new counterexamples $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{F}_1 \wedge \mathcal{H}$, $G' \vDash \mathcal{F}_1 \wedge \mathcal{H}$, $G \vDash \mathcal{F}_2$, and $G' \vDash \mathcal{F}_2$. We can continue this iteration by finding another constraint $\mathcal{F}_3$ not satisfied by all these graphs G that are part of a counterexamples and so on. If the procedure terminates, we will have a 1-inductive invariant $\bigwedge_{1 \le i \le n} \mathcal{F}_i$ under $\mathcal{H}$ for some value n.

However, this result is not always desirable: the technique changes the safety property and requires its verification in possible start graphs. Although the original property $\mathcal{F}_1$ will be a part of the new property, the result will be more complex and possibly less intuitive. Also, if a constraint $\neg\mathcal{F}_i$ does not exactly match the graphs in counterexamples – i.e. if there is a graph G with $G \nvDash \mathcal{F}_i$ that is not part of a counterexample $G \Rightarrow_{\mathcal{R}} G'$ – the new 1-inductive invariant may be too broad. Consider an extreme case: $\mathcal{F}_i$ = false is indeed violated by all relevant counterexamples G and its conjcuntion with other constraints is necessarily always a 1-inductive invaraint – but since no graph will satisfy it, it is useless as a safety property.

When implementing such a procedure, we also have to consider the problems of incompleteness and undecidability. This is particularly important because we need to construct the additional constraints $\mathcal{F}_i$. Given symbolic counterexamples in each iteration, we can use those to derive the respective constraint $\mathcal{F}_i$; however, if the approach is incomplete, $\neg\mathcal{F}_i$ may also describe non-counterexamples, which may lead to the problem sketched above. Finally, the procedure may not terminate for two reasons: the underlying problem of undecidability (even for the restricted formal model) or because there does not exist a 1-indcutive invariant $\bigwedge_{1 \le i \le n} \mathcal{F}_i$ under $\mathcal{H}$.

That said, we actually do apply a (different) type of iteration in the Seq-construction for the general and restricted approaches. Hence, there is a relation between analyzing singular transformations $G \Rightarrow_{\mathcal{R}} G'$ and k-induction:

Lemma 8.2 (iterative approach to k-induction). *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and let $\mathcal{H}$ and $\mathcal{F}_i$ with $1 \le i \le k$ be graph constraints such that:*

1. *For all i with $1 \le i \le k-1$ and all transformations $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$ and $G' \vDash \mathcal{H}$, we have $G' \nvDash \mathcal{F}_i \Rightarrow G \nvDash \mathcal{F}_1 \vee G \nvDash \mathcal{F}_{i+1}$.*
2. *For all transformations $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$ and $G' \vDash \mathcal{H}$, we have $G' \nvDash \mathcal{F}_k \Rightarrow G \nvDash \mathcal{F}_1$.*

Then, $\mathcal{F}_1$ is a k-inductive invariant of GTS under $\mathcal{H}$.

Proof. Consider a transformation sequence $G_0 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_k$ with $G_i \vDash \mathcal{H}$ for $i = 1, .., k$ and $G_i \vDash \mathcal{F}_1$ for $i = 1, .., k-1$. We need to show $G_k \vDash \mathcal{F}_1$.

We assume $G_k \nvDash \mathcal{F}_1$ and will first show $G_{k+1-i} \nvDash \mathcal{F}_i$ for $i = 1, .., k$ by induction over i.

Base case. For $i = 1$, $G_k \nvDash \mathcal{F}_1$ holds by assumption.

Inductive step. Consider $G_{k+1-(i+1)} \Rightarrow_{\mathcal{R}} G_{k+1-i}$. By inductive hypothesis, we have $G_{k+1-i} \nvDash \mathcal{F}_i$. By precondition (1), we get $G_{k+1-(i+1)} \nvDash \mathcal{F}_1 \vee G_{k+1-(i+1)} \nvDash \mathcal{F}_{i+1}$. Since, with $1 \le i \le k-1$, we have $G_{k+1-(i+1)} \vDash \mathcal{F}_1$, we get $G_{k+1-(i+1)} \nvDash \mathcal{F}_{i+1}$, concluding the inductive proof.

In particular, we have $G_1 \nvDash \mathcal{F}_k$ and by precondition (2), we get $G_0 \nvDash \mathcal{F}_1$, which is a contradiction. Hence, we have $G_k \vDash \mathcal{F}_1$. □

Again, we investigate counterexamples for a 1-inductive invaraint $\mathcal{F}_1$ under $\mathcal{H}$: transformations $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$, $G' \vDash \mathcal{H}$, $G \vDash \mathcal{F}_1$, and $G' \nvDash \mathcal{F}_1$. All such graphs G are described by a constraint $\mathcal{F}_2$; then, we check whether all transformations $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$ and $G' \vDash \mathcal{H}$ imply $G' \nvDash \mathcal{F}_2 \Rightarrow G \nvDash \mathcal{F}_1$. If not, we represent the respective graphs G in counterexamples by another constraint $\mathcal{F}_3$. We continue this iterative process until we reach a constraint $\mathcal{F}_k$ such that $G' \nvDash \mathcal{F}_k \Rightarrow G \nvDash \mathcal{F}_1$ in all transformations $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$ and $G' \vDash \mathcal{H}$.

Unlike the iterative approach that accumulates constraints and shows their invariance, the constraints $\mathcal{F}_i$ here (except for $\mathcal{F}_1$) are not verified as 1-inductive invariants. Given transformations $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$ and $G' \vDash \mathcal{H}$, instead of $G' \nvDash \mathcal{F}_i \Rightarrow G' \nvDash \mathcal{F}_i$ ($\mathcal{F}_i$ as a 1-inductive invariant under $\mathcal{H}$), we verify $G' \nvDash \mathcal{F}_i \Rightarrow G \nvDash \mathcal{F}_1$, meaning that violations of $\mathcal{F}_i$ are only reachable from violations of $\mathcal{F}_1$. As explained above, counterexamples are gathered in a constraint $\mathcal{F}_{i+1}$, which leads to $G' \nvDash \mathcal{F}_i \Rightarrow G \nvDash \mathcal{F}_1 \vee G \nvDash \mathcal{F}_{i+1}$ (under $\mathcal{H}$) as in condition (1) and $G' \nvDash \mathcal{F}_k \Rightarrow G \nvDash \mathcal{F}_1$ (2) as the termination criterion.

The condition $G' \nvDash \mathcal{F}_i \Rightarrow G \nvDash \mathcal{F}_1$ is equivalent to $G \vDash \mathcal{F}_1 \Rightarrow G' \vDash \mathcal{F}_i$. Its verification requires support for preconditions and postconditions: we need to be able to show that validity of a graph constraint $\mathcal{F}_i$ before rule application implies validity of another constraint $\mathcal{F}_i$ after rule application – provided $\mathcal{H}$ is satisfied both before and after rule application. This could be performed after lesser modifications to the existing approach for 1-inductive invariant checking [BBG+06, Dyc12] and its later incarnations *[1]*. Without a guaranteed constraint $\mathcal{H}$ (or with $\mathcal{H}$ = true), it can also be performed by techniques supporting verification with preconditions and postconditions, most notably the approach described and and implemented by Pennemann and Habel [HP09, Pen09]. With a non-trivial guaranteed constraint, the procedure becomes more involved. We will consider this in more detail in Section 8.2.

Intuitively, the idea of an implementation of Lemma 8.2 is to iteratively build symbolic transformation sequences leading to a violation and analyzing the sequence for earlier violations (of $\mathcal{F}_1$). Unsurprisingly, this is also the idea of the general (and restricted, see below) approach, which implements an iterative (but automated) procedure similar to the one sketched in Lemma 8.2. Given an s/t-pattern sequence $seq = src_1 \Rightarrow_b tar_1$ with $seq \in \mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}\wedge\mathcal{H}, \mathcal{F}\wedge\mathcal{H})$, the constraint $src_{1|\varnothing}$ describes exactly the graphs G such that there is a transformation $G \Rightarrow G'$ with $G \vDash \mathcal{H}$, $G' \vDash \mathcal{H}$, $G \vDash \mathcal{F}$, and $G' \nvDash \mathcal{F}$ – i.e. a counterexample. This constraint $src_{1|\varnothing}$, along with the respective constraints from other s/t-pattern sequences in $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$ is then paired with the right side of another rule in step $\mathrm{SC_k}$-1 – and so on in further iterations for higher values of k. Analysis for violations of $\mathcal{F}$ (or $\mathcal{F}_1$ in the lemma above) is performed by transferring $\mathcal{F}$ to each intermediate source pattern and, by Lemma 5.14 (p. 94) and Theorem T.2g (p. 96), by checking the sequence's satisfiability.

The situation is slightly different for the restricted approach. Given an s/t-pattern sequence $seq = src_1 \Rightarrow_b tar_1$ with $seq \in \mathrm{Seq}_1^r(\mathcal{R}, F)$ for a singular forbidden pattern F of $\mathcal{F}$, $src_{1|\varnothing}$ describes the set of graphs G such that there is a transformation $G \Rightarrow_b G'$ with $G' \vDash F$ (which implies $G' \nvDash \mathcal{F}$). As before, $src_{1|\varnothing}$ is used to construct $\mathrm{Seq}_k(\mathcal{R}, F)$ for subsequent values of k – unless $src_{1|\varnothing}$ is discarded. This may happen because it implies a forbidden or guaranteed pattern (by Theorem T.2r (p. 143)) or, if implication with composed patterns (Section 7.4) is used, because it implies $\neg\mathcal{F}\wedge\mathcal{H}$ (by Theorem T.2e-ai (p. 224)). However, since the approach is incomplete, the construction and theorem may fail to discard a number of false negatives, whose corresponding source patterns will then be used in the next iteration. In the restricted approach, this is partly addressed by forward propagation (Section 7.1). In a hypothetical implementation of Lemma 8.2 that would be based on previous implementations of 1-inductive invariant checking [BBG+06, Dyc12], this cannot be addressed without significant changes concerning the inclusion and analysis of forbidden and guaranteed patterns in source and target patterns. Thus, applying these techniques for 1-inductive invariant checking to perform k-induction – without optimizations specific to k-induction – risks higher numbers of false negatives or higher computational effort.

8.2. Verification of Graph Programs via Weakest Preconditions

Between 2006 and 2009, Pennemann, Habel, and Rensink [HPR06, Pen08a, Pen08b, HP09, Pen09] described an approach for the verification of graph programs with respect to a specification that consists of the program and its postcondition and precondition [Pen09]. Starting with the postcondition, the approach computes the weakest precondition such that after program execution, the postcondition is satisfied. If the specified precondition implies the computed precondition – which is verified – the program specification is correct: whenever the specified precondition is satisfied, program execution results in satisfaction of the postcondition [Pen09]. The approach has been implemented under the name Enforce.

Conditions are specified by graph constraints. There is work concerning the computation of weakest preconditions for *HR^* conditions* [Rad13], which are more expressive than nested graph constraints, but the current implementation focuses on the latter. Graph programs consist mainly of graph rules (with left nested application conditions) with control structures such as sequential execution, non-deterministic choice between rules, loops, and conditional execution [Pen09]. As such, conditions by themselves are equivalent to properties as described in our general formal model, but more expressive than (composed) graph patterns in the restricted formal model. While graph transformation systems without control structures (as in the general and restricted formal models) are already turing-complete [HP01], graph programs allow the use of constructs typical for imperative programming and, as such, allow a much more direct specification of the respective system's behavior.

The approach contains two main components: the computation of the weakest precondition and the implication check. The former is performed by constructions identical or similar to the Shift- and L-constructions. The latter is essentially the probem of implication of graph constraints, which is undecidable for nested graph constraints in general. Still, the approach and implementation attempt to provide an answer with a timeout ensuring termination at the cost of potentially indecisive results. It is based on the satisfiability solver SeekSat and the theorem prover ProCon [Pen09]: SeekSat [Pen08a] is able to find satisfying graphs for satisfiable graph constraints in finite time [Pen09], ProCon implements a calculus using deduction rules on graph constraints [Pen08b].

Given a graph program P and pre- and postconditions $\mathcal{C}$ and $\mathcal{C}'$, the respective specification is denoted as $\{\mathcal{C}\}P\{\mathcal{C}'\}$ [Pen09]. If the program (specification) is correct and given graphs G and G' where G' is the result of executing P on G, $G \vDash \mathcal{C}$ implies $G \vDash \mathcal{C}'$. We can apply this approach to verify a 1-inductive invariant $\mathcal{F}$ for a graph transformation system $GTS = (TG, \mathcal{R})$ under a trivial guaranteed constraint $\mathcal{H} = \text{true}$: if $\{\mathcal{F}\}\mathcal{R}\{\mathcal{F}\}$ is correct, then $\mathcal{F}$ is a 1-inductive invariant of GTS. Here, $\mathcal{R}$ is a graph program that applies a non-deterministically chosen rule from the set $\mathcal{R}$; then, for all graph transformations $G \Rightarrow_{\mathcal{R}} G'$, $G \vDash \mathcal{F}$ implies $G' \vDash \mathcal{F}$ as required. In fact, several case studies used to evaluate Enforce have programs specified with 1-inductive invariants [Pen09]. If a non-trivial guaranteed constraint is involved, the approach can also be applied *[1]*:

Lemma 8.3 (1-induction with weakest preconditions)**.** *Let $GTS = (TG, \mathcal{R})$ be a graph transformation system and $\mathcal{F}$ and $\mathcal{H}$ be graph constraints. $\mathcal{F}$ is a 1-inductive invariant of GTS under $\mathcal{H}$, if the program specification $\{\mathcal{F} \wedge \mathcal{H}\}\mathcal{R}\{\mathcal{F} \vee \neg\mathcal{H}\}$ is correct.*

Proof. Correctness of $\{\mathcal{F} \wedge \mathcal{H}\}\mathcal{R}\{\mathcal{F} \vee \neg\mathcal{H}\}$ implies that for each rule $b \in \mathcal{R}$, we have

$$\forall G, G'((G \Rightarrow_b G') \Rightarrow (G \vDash \mathcal{F} \wedge \mathcal{H} \Rightarrow G' \vDash \mathcal{F} \vee \neg\mathcal{H})),$$

which is equivalent to

$$\forall G, G'((G \Rightarrow_b G' \wedge G \vDash \mathcal{H} \wedge G' \vDash \mathcal{H}) \Rightarrow (G \vDash \mathcal{F} \Rightarrow G' \vDash \mathcal{F})),$$

which is the definition of a 1-inductive invariant $\mathcal{F}$ under a constraint $\mathcal{H}$ (Definition 4.4). □

While applicable, the approach and implementation may run into problems for larger conditions: in contrast to the restricted approach described in this thesis, the general nature of the Enforce implementation puts less emphasis on optimizations for a restricted fragment of constraints and application conditions. As a result, Enforce is impractical for certain scenarios *[1]*. On the other hand, it can be applied to systems whose specifications require graph constraints beyond the restricted formal model described in this thesis. We will reexamine differences in performance and specifications in Chapter 9, Section 9.2.

The situation becomes more involved if we attempt to perform k-inductive invariant checking. Since we can apply Enforce to verify singular applications of graph rules, we can use the technique sketched in Section 8.1 (Lemma 8.2 (p. 233)): iterative analysis of singular applications of transformation rules to reason about k-inductive invariants. Part of this scheme was to verify that for a set of rules $\mathcal{R}$ and all graphs $G \Rightarrow_{\mathcal{R}} G'$ with $G \vDash \mathcal{H}$ and $G' \vDash \mathcal{H}$, $G \vDash \mathcal{F}$ implies $G' \vDash \mathcal{F}'$ for graph constraints $\mathcal{F}$ and $\mathcal{F}'$. This is equivalent to showing correctness of $\{\mathcal{F} \wedge \mathcal{H}\}\mathcal{R}\{\mathcal{F}' \vee \neg\mathcal{H}\}$. We would again need symbolic counterexamples from each iteration as input for the next iteration. Those are not currently provided by Enforce; however, the necessary algorithms are already part of the tool and could be reused. The same applies if we were to accumulate counterexamples in new constraints $\mathcal{F}_i$ until the conjunction $\bigwedge_{1 \leq i \leq n} \mathcal{F}_i$ is a 1-inductive invariant under $\mathcal{H}$. In both cases, Enforce could be modified to allow automation of this iterative approach – or an automated algorithm could repeatedly and iteratively call Enforce with each iteration's parameters. As with 1-inductive invariant checking, however, performance will be a limiting factor.

Unfortunately, performing general k-induction with Enforce directly is not possible without changing the graph program in question. In particular, correctness of the graph program $\{\mathcal{F} \wedge \mathcal{H}\}\mathcal{R}^k\{\mathcal{F} \vee \neg\mathcal{H}\}$ – with $\mathcal{R}^k$ a graph program of k subsequent applications of rules chosen nondeterministically from $\mathcal{R}$ – does not mean that $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$. Instead, it proves that for all graphs G, G' with $G \Rightarrow_{\mathcal{R}}^k G'$, $G \vDash \mathcal{F} \wedge \mathcal{H}$ implies $G' \vDash \mathcal{F} \vee G' \nvDash \mathcal{H}$. Verification of a k-inductive invariant would also require checking intermediate graphs in the transformation sequence $G \Rightarrow_{\mathcal{R}}^k G'$.

That said, the inclusion of the analysis of intermediate graphs is possible by inserting special graph rules into the graph program. Consider a graph rule $\mathsf{addC} = \langle(\varnothing \hookleftarrow \varnothing \hookrightarrow \varnothing), \mathcal{F} \wedge \mathcal{H}, \text{true}\rangle$. This rule is applicable to all graphs that satisfy $\mathcal{F} \wedge \mathcal{H}$ and has no effect on the graph. If we then change the graph program to $\{\mathcal{F} \wedge \mathcal{H}\}\{\mathcal{R}; \mathsf{addC}\}^{k-1}\mathcal{R}\{\mathcal{F} \vee \neg\mathcal{H}\}$, the required application of addC after each regular rule application (from $\mathcal{R}$) incorporates satisfiability of $\mathcal{F} \wedge \mathcal{H}$ in intermediate steps, allowing us to perform k-inductive invariant checking. Again, performance will be a limiting factor – in particular, because shifting the entire constraint $\mathcal{F} \wedge \mathcal{H}$ to each intermediate condition is costly. This is investigated further in Chapter 9, Section 9.3.3.

The situation is different for k−1-bounded backward model checking with a graph transformation system $GTS = (TG, \mathcal{R})$, a start configuration constraint $\mathcal{S}$, a safety property $\mathcal{F}$, and a trivial guaranteed constraint $\mathcal{H} = \text{true}$. Correctness of all programs

$$\begin{array}{ll} \{\mathcal{S}\}\mathcal{R}^1 & \{\mathcal{F}\}, \\ \{\mathcal{S}\}\mathcal{R}^2 & \{\mathcal{F}\}, \\ \dots\ \dots & \dots, \\ \{\mathcal{S}\}\mathcal{R}^{k-1} & \{\mathcal{F}\} \end{array}$$

is equivalent to the validity of $\mathcal{F}$ in the k−1-bounded state space of all induced graph grammars in IND$(GTS, \mathcal{S})$. Here, we do not need to transfer additional constraint to intermediate conditions: $\mathcal{F}$ does not need to be considered (cf. Theorems T.2g and T.2r) and $\mathcal{H}$ is trivial. If a non-trivial guaranteed constraint is involved, we need to apply a variant of the addC rule introduced above: given $\mathsf{addH} = \langle(\varnothing \hookleftarrow \varnothing \hookrightarrow \varnothing), \mathcal{H}, \text{true}\rangle$, we have to verify all programs

$$\begin{array}{lll} \{\mathcal{S} \wedge \mathcal{H}\} & \mathcal{R} & \{\mathcal{F} \vee \neg\mathcal{H}\}, \\ \{\mathcal{S} \wedge \mathcal{H}\} & \{\mathcal{R}, \{\mathsf{addH}, \mathcal{R}\}^1\} & \{\mathcal{F} \vee \neg\mathcal{H}\}, \\ \dots & \dots & \dots, \\ \{\mathcal{S} \wedge \mathcal{H}\} & \{\mathcal{R}, \{\mathsf{addH}, \mathcal{R}\}^{k-2}\} & \{\mathcal{F} \vee \neg\mathcal{H}\}. \end{array}$$

Of course, Enforce's primary purpose of verifying graph programs does not rely on induction (or bounded backward model checking). It is meant to analyze graph programs with respect to preconditions and postconditions, not necessarily with respect to inductive invariants. In some scenarios, the former perspective is more important. For instance, Enforce can verify a graph program's (system's) capability to recover from a safety violation in a certain number of steps or by a certain sequence of rule applications. Given a safety property $\mathcal{F}$, this means verifying $\{\neg\mathcal{F}\}P\{\mathcal{F}\}$: starting in a graph G with $G \nvDash \mathcal{F}$, application of P always results in a graph G with $G \vDash \mathcal{F}$. Similarly, given a set of rules $\mathcal{R}$, correctness of $\{\neg\mathcal{F}\}\mathcal{R}^k\{\mathcal{F}\}$ means that after k applications of rules in $\mathcal{R}$, safety is restored. These are system properties that cannot be shown with either the general or restricted approach (without significant changes) described in this thesis.

8.3. Verification of Infinite-State Graph Transformation Systems via Abstraction

Between 2011 and 2015, Steenken, Wehrheim, and Wonisch [SW11, SWW11, Ste15] presented an approach and implementation focused on verification of graph transformation systems with infinite state spaces[1] via finite-state abstraction, shape analysis, and model checking for state properties. The general idea is to abstract from a graph transformation system and its state space by a *shape transformation system* whose rules operate on shapes instead of specific graphs [Ste15]; this notion has previously been described in and is based on a number of publications by several authors, including Boneva, Distefano, Kreiker, Kurbán, Rensink, and Zambon [BBKR08, BKK+12, RD06]. A *shape* may encode an infinite number of graphs; it may contain *summary nodes*, which represent any positive number of nodes and 1/2-*edges*, which specify that the respective edge may or may not exist [Ste15]. Additional *shape constraints* – first-order logic formulas over node and edge predicates – may be used to further restrict shapes. When rules of a shape transformation system are applied, shapes are *materialized*: summary nodes and 1/2-edges matched by the respective rule are required to be present, meaning that the materialized shape contains them as regular nodes and edges in addition to the summary nodes and 1/2-edges [Ste15, SWW11, SW11]. To preserve a uniform level of abstraction (to the extent possible), abstraction schemes may be applied to shapes after materialization [Ste15].

This form of abstraction may significantly reduce the size of a transformation system's state space's representation and, for infinite state spaces, may provide a finite encoding. During (lazy) construction of this representation, the approach and implementation searches for traces from a transformation system's initial graph to a forbidden pattern. Such a pattern is described by a subgraph that should not occur in any safe system state. The approach and abstraction is sound: if no occurrene of the forbidden pattern can be found in the state space of the shape transformation system, the pattern will not occur in the (non-abstracted) graph transformation system's state space either. The reverse, however, does not necessarily hold.

[1] Note that the approach defines a graph transformation system as a pair of a rule set and an initial state; then, a graph transformation system has a state space. This thesis uses a different formalism: a graph grammar, not a graph transformation system, contains an initial state and hence, has a state space.

Given the expressive power of formulas in shape constraints, the approach also inherits the underlying undecidability of first-order logic [Ste15]. This manifests in incompleteness; specifically, the possibility of *spurious counterexamples*. To alleviate this, abstract error traces are analyzed for their feasibility, i.e. validated with respect to the existence of a corresponding concrete error trace from the initial state to the forbidden pattern.

The approach is implemented as an automated tool and has been evaluated for several case studies [Ste15]: operations on linear lists, computer networks protected by a firewall and its protocol [KK06, ZR12], and process scheduling [Won10].

Because of its design, the approach cannot be used (and is not intended) to verify 1-inductive invariants or k-inductive invariants. However, it is capable of verifying operational invariants, which usually is the ultimate purpose of applying 1-induction or k-induction (cf. Lemma L.1 (p. 65)). Since it applies model checking on the (abstracted) state space, lack of context information is expected to be less of a problem, and reachability is considered by the analysis. On the other hand, the approach relies on state spaces that can be represented in a finite fashion and feasible size. While applicable to a number of meaningful examples, the current implementation only supports verification for a singular initial state, a singular forbidden pattern, and rules without application conditions [Ste15]. These appear to be restrictions of the implementation and less of the underlying theory [Ste15]; in particular, extension by negative application conditions is claimed to require litte additional effort [Ste15]. Similarly, a system's initial graph could be changed to allow an initial shape, which would support verification of an infinite number of initial graphs in one verification run. Using forbidden shapes instead of forbidden patterns (or error patterns) could also be achieved, although it would require more effort [Ste15].

A error pattern or forbidden pattern in the form of a graph P in this approach corresponds to an existential condition $\exists(i_P : \varnothing \hookrightarrow P)$ in the formalism used in this thesis, i.e. to a forbidden pattern with a trivial composed negative application condition true. Hence, we can attempt to mimic the verification result by the approach by Steenken, Wehrheim, and Wonisch to some degree: using the restricted approach and implementation proposed in this thesis, we can attempt to verify a forbidden pattern as a k-inductive invariant and establish its validity in a singular graph grammar's $k-1$-bounded state space (Lemmas L.1 (p. 65) and L.2 (p. 66)). If successful, the property can be confirmed as an operational invariant without having to compute the state space (or its abstraction). If verification as a k-inductive invariant fails, however, we do not usually get an error trace from the graph grammar's initial state – in contrast to the approach by Steenken, Wehrheim, and Wonisch (and to $k-1$-bounded backward model checking). Also, even if the pattern is not a k-inductive invariant, it may still be an operational invariant if the initial state(s) is suitably chosen. This, too, is not considered by the approach discussed in this thesis.

In summary, both approaches follow different ideas; each will be better suited than the other in specific application scenarios. While the approach by Steenken, Wehrheim, and Wonisch is likely to offer more intuitive and meaningful counterexamples and provides a higher degree of completeness, the restriced approach discussed in this thesis offers support for more expressive specifications; furthermore, feasibility of verification may be less dependent on the size of state spaces. Given system specifications supported by both approaches, there is the possibility of combining them: if verification with k-inductive invariants fails, abstraction and model checking may still prove operational invariance – or provide a meaningful error trace. In Section 9.4, we will discuss a case study used to evaluate verification by abstraction and how it relates to inductive invariant checking.

9. Application and Evaluation

In this chapter, we will apply the restricted approach to verification with k-inductive invariants to a number of case studies and compare the case studies and results to examples and tools described in related work.

The restricted approach was implemented as a tool prototype in a series of plugins for the Eclipse[1] development environment. The tool first appeared [BBG⁺06] as part of the Fujaba tool suite [NNZ00]. A later version was decoupled from Fujaba and migrated to a new version based on the Eclipse Modeling Framework[2]. At first, its formal model supported only negative application conditions containing (at most) a node and an edge; later, the tool supported 1-inductive invariant checking with respect to most of the restricted formal model [Dyc12]. However, support for guaranteed patterns was limited and had no formal description. Of the extensions described in Chapter 7, only rule priorities [BBG⁺06, Dyc12] were already implemented (for 1-induction).

In the context of this thesis, the implementation was extended to support k-inductive invariant checking (for the restricted formal model, including guaranteed patterns). Rule applicability conditions (steps SC_1-3 and SC_k-3 of the Seq-constructions) were also added. Forward propagation (Section 7.1), partial negative application conditions (Section 7.3), and implication of composed graph patterns (Section 7.4) were implemented; handling of rule priorities (Section 7.2) was extended to k-inductive invariant checking. These four extensions can be combined with one exception: currently, rule priorities are not taken into account for context generation and context reduction.

The procedure of $k-1$-bounded backward model checking was also implemented. Its main purpose is to verify the base case of our inductive argument, with k-inductive invariant checking as the inductive step. The implementation includes support for the four extensions listed above. Previously, the base case was not explicitly verified. Establishing a 1-inductive invariant $\mathcal{F}$ for a graph transformation system GTS meant that graph grammars with an error-free initial state were safe: for all graph grammars $GG = (GTS, G_0)$ with $G_0 \models \mathcal{F}$, all graphs in their state spaces would satisfy $\mathcal{F}$ (by Lemma 3.4 (p. 50)). Now, as discussed in Chapter 4, we may specify a start configuration constraint $\mathcal{S}$, which, in the restricted approach, is required to be a composed (start configuration) pattern. Thus, verification of the base case (as in Lemma L.1 (p. 65)) requires either validity of $\mathcal{F}$ in the $k-1$-bounded state space (if $k > 1$) or $\mathcal{S} \models \mathcal{F}$ (if $k = 1$). The first case can be verified by $k-1$-bounded backward model checking. The second case can be checked by implication with composed graph patterns, but will not be the focus here.

Outline. This chapter is structured as follows:

Section 9.1 explains the architecture and implementation of the tool and approach. It also describes configuration options and explains how experiments were conducted and how their results should be interpreted.

Section 9.2 discusses the case study about behavior preservation for relational model transformations, which was introduced in earlier work [GL12] and extended in more recent publications *[6, 5]*. Experiments for this case study were performed with both 1-inductive invariant checking and Enforce (SeekSat/ProCon).

[1] https://www.eclipse.org

[2] https://www.eclipse.org/modeling/emf/

Section 9.3 picks up the running example of this thesis: the protocol for behavior of autonomous shuttles on switches. For this example, we consider k-induction with k ranging from 1 to 5. Variations and fragments of this case study will also be used to demonstrate the effect of partial negative application conditions and to investigate to what extend k-induction can be performed by either 1-induction or application of Enforce.

Section 9.4 discusses the example of a list with rules to add and remove elements [Ste15]. It demonstrates a limitation of inductive invariant checking and how this limitation can be partly addressed.

Section 9.5 explains why the existence of multiple behavioral entities with concurrent behavior in graph grammars may lead to problems for $k \geq 2$ and how we may attempt to address the issue.

Finally, Section 9.6 summarizes the results of this chapter.

9.1. Architecture and Configuration

The tool follows a pipe-and-filter architecture. Computations are performed by filters, which are arranged sequentially and are connected by pipes. A task, such as verification of a k-inductive invariant, is performed by a filter chain, which contains (in sequence)

- optional Preprocessors, which may perform preparing tasks on the data (sequentially),
- a Producer, which partitions the data as required and passes the pieces to the first filter,
- a sequence of Filters, which perform computations on the pieces of data, and
- finally, a Consumer, which is responsible for processing the pieces of data received from the last filter and displaying the results.

The composition and configuration of a filter chain determines its task and which extensions, if any, should be used. For instance, the tool has predefined filter chains for k-inductive invariant checking with or without forward propagation for values of k ranging from 1 to 6. The same applies to $k-1$-bounded backward model checking with k ranging from 2 to 6. New filter chains – i.e. new combinations of a producer, consumer, filters, and their configuration parameters – can be put together by a user through the UI. New filters (or producers, consumers, or preoprocessing components) for tasks or algorithms not required by the tool in its current state would have to be implemented first.

The filters currently available cover the implementation of the algorithms required for the restricted approach and its extensions. For example, steps SC_1-2 and SC_k-2 of the Seq-construction are implemented by the so-called RuleApplicationFilter. Its configuration options determine which version of the Seq-construction is used: Theorem T.1r (p. 130) – the restricted approach – or Theorem 7.28 (p. 199) – the restricted approach with partial negative application conditions. Similarly, forward propagation is implemented in the ContextPropagationFilter. For k-inductive invariant checking and $k-1$-bounded backward model checking, each filter typically operates on one s/t-pattern sequences at a time. Any s/t-pattern sequence reaching the consumer is saved in a file as a counterexample. Such a counterexample can be viewed in a tree editor and contains the sequence's (reduced) source and target patterns and the rules connecting them.

The input of a filter chain comes from a model file where system elements can be specified in a tree editor that conforms to a predefined EMF metamodel for type graphs, graphs, graph rules, application conditions, and graph constraints. The following elements need to be specified:

- the type graph TG,
- the graph rules $\mathcal{R}$ (with priorities, if required),

- a composed forbidden pattern $\mathcal{F}$,
- a composed guaranteed pattern, if required $\mathcal{H}$, and
- a composed start configuration pattern, if required $\mathcal{S}$.

Through appropriately combined filter chains and configuration options, the current implementation allows to perfom k-inductive invariant checking (k-invcheck) and $k-1$-bounded backward model checking ($k-1$-modelcheck) in the sense of the restricted approach (Theorems T.2r (p. 143) and T.3r (p. 149)). With respect to extensions, both techniques can be executed

- with or without *forward propagation* (prop.) in the sense of Theorem T.2e-fp (p. 177),
- with or without *rule priorities* (prio.) in the sense of Theorem T.2e-rp (p. 184),
- with or without *implication with composed patterns* (impl.) in the sense of Theorem T.2e-ai (p. 224), and
- with or without *partial negative application conditions* (partial) in the sense of Theorem T.2e-pn (p. 211),

and with or without preprocessing (prepr.) to reduce composed forbidden patterns in the sense of Theorem 7.40 (p. 225): if the conjunction of the composed guaranteed pattern and a subset of forbidden patterns implies all remaining forbidden pattern, this remaining set does not have to be verified. For purposes of bounded backward model checking, there is also a preprocessor available to compute whether a composed start configuration pattern $\mathcal{S}$ implies all individual forbidden patterns F_i; if this is not the case, the required forbidden patterns can be added to the composed start configuration pattern. Combinations of extensions follow Theorem T.2e (p. 228).

Available configuration options include the following, which are mostly aimed at improving performance by skipping computations not required for the respective system:

- *Rule applicability conditions* usually created by steps SC_1-3 and SC_k-3 will be omitted from an *s*/*t*-pattern sequences, unless the option (appl.) is chosen.
- *Forward propagation* (prop.), if used, will skip propagation of composed negative application conditions by default, instead propagating only positive context in forward direction, unless the option prop.+NACs is chosen.
- Filter chains can be configured to abort computation as soon as one counterexample is found (stop).
- Output of counterexamples to files can be skipped (nf).
- The threshold value for context generation (which enforces termination for implication with composed patterns) can be changed.

For each experiment, the corresponding list of results contains

- the *time*, given in seconds,
- the number of counterexamples *# ce*,
- and the *result* of the verification, which is one of the following:

  T – if the property was successfully shown (a true positive),
  F – if there are only valid counterexamples (true negatives),
  FN – if there are only spurious counterexamples (false negatives),
  F+FN – if there are both valid and spurious counterexamples,
  F/FN – if the counterexamples may contain both valid and spurious counterexamples (but can also contain just one type),
  TmOut – if verification exceeded a time limit of one hour, or

MErr – if verification encountered an OutOfMemoryError, such as when exceeding Java heap space.

In general, any counterexample given by the tool may either be a false negative or a true negative (i.e. F or FN). If a counterexample disappears once certain extensions (forward propagation or implication with composed patterns) are used, it is a false negative. We use this insight to classify counterexamples as false negatives (FN) in the verification results documented and discussed in this chapter. We can also classify some counterexamples as true negatives (F) because of the form of the respective systems: for instance, if all guaranteed and forbidden patterns only have trivial composed negative application conditions, only one class of false negatives can occur – and this class can be eliminated by using forward propagation. However, this is not part of an automatic classification by the tool; strictly speaking, it only classifies results as T or F/FN.

All experiments were executed on a laptop computer with an Intel Core i7-2640M processor with two cores at 2,8 GHz and running Windows 7 Professional (Service Pack 1, 64 bit), Eclipse 4.5.1 (Mars.1), and Java 8. While the computer had 8 GB of main memory, the limit on Java heap space was set to 4 GB. All times are specified in seconds unless otherwise noted. Values under a second were not distinguished, values over five minutes were rounded down to the next minute. As mentioned above, verification tasks taking longer than one hour were aborted.

In order to provide a rough classification of verification tasks with respect to complexity, we will use a metric adapted from earlier work *[1]*: for each pair of a rule and a forbidden pattern, we will determine the number of overlappings of nodes in the pattern – including its composed negative application condition – and the right rule side. As established in Chapter 7, Section 7.3, that number is

$$ol(n,m) = \sum_{i=0}^{m} \binom{m}{i}\binom{n}{i} i!$$

for two graphs consisting of n and m nodes of the same type. Now, we consider a rule set $\mathcal{R}$ with right rule sides R, a type graph TG with a node type set V_{TG}, and a composed forbidden pattern $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ with $F_i = \exists(i_{P_i^F}, ac_i)$. Furthermore, let t_v^i be the number of nodes of type v (with $v \in T_{TG}$) in the existential condition's graph P_i^F of a pattern F_i. Likewise, t_v^b is the number of nodes of type v in the right side of a rule b. Then,

$$\begin{aligned} cy^{+}(\mathcal{F},\mathcal{R}) &= \sum_{(i,b)\in(I\times\mathcal{R})} \left(\prod_{v \in V_{TG}} ol(t_v^b, t_v^i) \right) \\ &= \sum_{(i,b)\in(I\times\mathcal{R})} \left(\prod_{v \in V_{TG}} \sum_{i=0}^{t_v^i} \binom{t_v^i}{i}\binom{t_v^b}{i} i! \right) \end{aligned}$$

is the sum of such overlappings (of nodes) for all pairs of forbidden patterns and right rule sides. More specifically, it is the minimum number of target patterns created by step SC$_1$-1 in $\mathrm{Seq}_1^r(\mathcal{R}, \neg\mathcal{F}) = \bigcup_{i \in I} \mathrm{Seq}_1^r(\mathcal{R}, F_i)$. If the pattern and right rule side do not contain any edges, it is the exact amount of target patterns created in these steps. Each target pattern thusly created may spawn an s/t-pattern sequence (of length 1) that needs to be analyzed – and that may then spawn additional s/t-pattern sequences for values $k > 1$.

Furthermore, let $F_i = \exists(i_{P_i^F}, ac_i)$ (as above) and $ac_i = \bigwedge_{j \in J_i} \neg\exists(x_{ij} : P_i^F \hookrightarrow X_{ij})$. Then, given t_v^{ij} as the number of nodes of type v in X_{ij},

$$cy^{-}(\mathcal{F},\mathcal{R}) = \sum_{(i,b)\in(I\times\mathcal{R})} \left(\sum_{j \in J_i} \left(\prod_{v \in V_{TG}} \sum_{i=0}^{t_v^{ij}} \binom{t_v^{ij}}{i}\binom{t_v^b}{i} i! \right) \right)$$

is the sum (over all pairs of rules and forbidden patterns) of the maxima of negative application conditions (considering nodes only) in a target pattern created from the respective pair. Edges are disregarded because computation of overlappings is more complex for nodes than for edges – often, overlappings of edges can be inferred by nodes. Negative application conditions are costly to compute (by the Shift-construction), have to be propagated along rule applications, and play an important role in discarding counterexamples via implication of patterns. As such, while cy^+ provides an estimate of the number of target pattern initially created (i.e. for $k = 1$), cy^- roughly describes the effort of creating and handling the target patterns with the highest number of negative application conditions. Of course, many target patterns will have a much lower number of such conditions. Furthermore, cy^+ and cy^- do not provide information about the size of patterns or negative application conditions in terms of the number of nodes and edges.

Note that both values only provide some indication as to the system's potential complexity, which may not correspond to the complexity and required runtime of the verification. For example, a combination of large patterns with large right rule sides will lead to a high value of cy^+ – but if all potential counterexamples are quickly discarded because of a small guaranteed pattern, verification will not require much computational effort. Likewise, certain optimizations – such as partial negative application conditions or preprocessing forbidden patterns in the sense of Theorem 7.40 (p. 225) – may lead to fewer and faster computations than expected. Also, the size of k in k-inductive invariant checking will not factor into the complexity. Still, the values provides some idea of the scope of the specified system and may highlight the effects of optimizations and extensions in the implementation of the restricted approach.

9.2. Case Study: Behavior Preservation of Model Transformations

This case study consists of six example systems, two of which were briefly sketched in Chapter 1, Examples 1.2 (p. 6) and 1.3 (p. 7). The systems were used to describe and verify behavior preservation for relational model transformations (specifically, triple graph grammars). The goal of the approach, originally introduced in 2012 [GL12], was to verify behavioral equivalence (and later, refinement [*6*, *5*]) for all possible source and target models – an infinite amount. Hence, verification needed to be performed symbolically on the transformation level – for all pairs of source and target models – instead of the instance level, where only individual pairs of a source and a target model can be analyzed.

Details of the approach can be found in the original source [GL12] and subsequent publications [*6*, *5*]. In short, there are two phases: model transformation and model semantics. A model transformation is specified by a triple graph grammar (TGG) [Sch94]; a triple graph grammar contains an axiom and a set of triple graph rules, which essentially are (typed) graph transformation rules. It is also equipped with a constraint restricting its state space. The triple graph grammar creates source and target models along each other and thusly generates all pairs of source and target models, i.e. all model transformation instances. Of course, the grammar may only create source and target models conforming to the source and target modeling languages, which are specified by type graphs and graph constraints.

Model semantics are also specified by graph transformation rules and further restricted by graph constraints. They are typed over the runtime source and target modeling languages, which allow for the occurrence of additional runtime elements not present (and not relevant) for the model transformation phase (above). In order to specify and verify behavioral equivalence between the semantics of two models, rules in source semantics have equivalent rules in the target semantics (and, since they are equivalent, vice versa). Behavioral equivalence – in the sense of the approach [*6*] – then requires bisimilarity [Mil89]: if every rule application in the source model can be followed by an equivalent rule application in the target model, and vice versa, the two models are behaviorally equivalent. If this holds for each pair of source and target models, the model transformation as such is behavior preserving.

In order to verify bisimilarity, the approach describes the construction of a *bisimulation constraint* – a graph constraint. Given the constraint's validity in the triple graph grammar's axiom, the approach applies 1-inductive invariant checking: it is shown *[5, 6]* that the invariance of the bisimulation constraint for the triple graph grammar and the combined model semantics implies the existence of a bisimulation relation for the labeled transition systems induced by the model semantics. In its first version [GL12], the approach imposed limitations on the nature of model semantics; in particular, no non-determinism was allowed when rules were applied. In the extended version *[6, 5]*, this restriction was dropped; furthermore, verification of behavioral refinement via a simulation relation was introduced.

All examples used in the original [GL12] and follow-up publications *[6, 5, 7]* describe transformations and semantics of sequence charts and communicating automata. Each example consists of two phases: model transformation and combined model semantics. Both require verification of the invariance of a composed forbidden pattern under a composed guaranteed pattern. For the combined model semantics, that composed forbidden pattern is the bisimulation or simulation constraint; for the model transformation, it is a conjunction of the bisimulation (or simulation) constraint and an additional model transformation constraint derived from the former. There are three examples:

equiv-s: This first example [GL12] allows only source and target models with one lifeline and one automata, respectively; it is shown that the model transformation is indeed behavior preserving in an equivalent manner. Its two phases are equiv-s-trans and equiv-s-sem. All system elements are listed in Section C.2.1 of Appendix C.

equiv: This second, more permissive example *[6, 5]* allows an arbitrary number of lifelines in sequence charts and an arbitrary number of automata in systems of communicating automata. The model transformation was also shown to preserve behavior equivalently. The two phases are equiv-trans and equiv-sem. All system elements are listed in Section C.2.2 of Appendix C.

refine: Finally, the third example is similar to the second, but systems with communicating automata may have additional internal behavior. Then, the transformation from communicating automata to sequence charts is behavior preserving in a refining manner: any behavior in sequence charts is reflected in the automata – but, given the internal behavior in automata, the reverse does not hold. Again, two phases refine-trans and refine-sem are involved. All system elements are listed in Section C.2.3 of Appendix C.

In all cases, we apply 1-inductive invariant checking for a graph transformation system under a composed guaranteed pattern. These composed guaranteed patterns contain mostly type graph restrictions, which, in the checking tool, cannot (yet) be encoded directly in the type graph. They also contain some assurances that follow from the two different phases of the approach and the nature of model semantics rules and model transformation rules. For instance, model semantics are only allowed to delete or create elements designated as runtime elements. Hence, invariants defined over the source and target modeling languages – as opposed to the runtime source and target modeling languages – and established for the model transformation rules will also be invariant for model semantics.

In general, constraints and rules in the approach do not necessarily conform to the restricted formal model. In the examples used here, most do. For some rules and constraints, a simplification scheme for negative application conditions had to be used *[5]* to equivalently fit them to the restricted formal model.

We do not apply bounded backward model checking for these examples. The base case for the model semantics phase is established by the model transformation phase; the base case

Table 9.1. – Verification problems and results for behavior preservation

			Enforce SeekSat/Procon		1-invcheck								
					-			Impl.			Impl., prepr.		
Example ($|\mathcal{R}|$/ $|\mathcal{F}|$/ $|\mathcal{H}|$)	cy^+	cy^-	time (s)	result	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result
equiv-s-trans-init/init-mt (2/2/32)	12	36	1	T	< 1	0	T	< 1	0	T	< 1	0	T
equiv-s-trans-msg/ts-mt (2/4/32)	84	980	266	T	< 1	0	T	< 1	0	T	< 1	0	T
equiv-s-trans-d n (2/18/32)	184	132	79	T	< 1	4	FN	< 1	0	T	< 1	0	T
equiv-s-trans (2/24/32)	280	1148	5 min	T	< 1	4	FN	< 1	0	T	< 1	0	T
equiv-s-sem-rt (3/2/48)	16	16	50	T	< 1	6	FN	< 1	0	T	< 1	0	T
equiv-s-sem-init/init (3/2/48)	16	16	23 min	T	< 1	0	T	< 1	0	T	< 1	0	T
equiv-s-sem-msg/ts (3/4/48)	96	96	> 1 h	TmOut	< 1	20	FN	3	0	T	< 1	0	T
equiv-s-sem (3/8/48)	128	128	> 1 h	TmOut	< 1	26	FN	3	0	T	< 1	0	T
equiv-trans-init/init-mt (2/2/28)	38	38		MErr	< 1	0	T	< 1	0	T	< 1	0	T
equiv-trans-msg-mt (2/1/28)	2941	20497		MErr	91	0	T	90	0	T	89	0	T
equiv-trans-com-mt (2/1/28)	20497	2941		MErr	31	0	T	32	0	T	31	0	T
equiv-trans-d n (2/16/28)	23822	23490		MErr	139	2	FN	124	0	T	< 1	0	T
equiv-trans (2/22/28)	47298	46966		MErr	5 min	2	FN	5 min	0	T	124	0	T
equiv-sem-rt (2/2/44)	14	14		MErr	< 1	2	FN	< 1	0	T	< 1	0	T
equiv-sem-init/init (2/2/44)	38	38		MErr	< 1	2	FN	< 1	0	T	< 1	0	T
equiv-sem-send (2/1/44)	2941	20497		MErr	16 min	19	MErr	26 min	0	T	< 1	0	T
equiv-sem-fire (2/1/44)	20497	2941		MErr	199	36	FN	5 min	0	T	< 1	0	T
equiv-sem (2/6/44)	23490	23490		MErr	18 min	59	MErr	29 min	0	T	< 1	0	T
refine-trans-init-mt (2/1/26)	25	49		MErr	< 1	0	T	< 1	0	T	< 1	0	T
refine-trans-msg-mt (2/1/26)	2955	61461		MErr	16 min	0	T	17 min	0	T	16 min	0	T
refine-trans-d n (2/15/26)	3633	61527		MErr	18 min	1	FN	17 min	0	T	< 1	0	T
refine-trans (2/17/26)	6613	123037		MErr	34 min	1	FN	34 min	0	T	17 min	0	T
refine-sem-rt (2/1/40)	7	7		MErr	< 1	1	FN	< 1	0	T	< 1	0	T
refine-sem-init (2/1/40)	19	19		MErr	< 1	1	FN	< 1	0	T	< 1	0	T
refine-sem-send (2/1/40)	2941	20497		MErr	20 min	19	MErr	28 min	0	T	< 1	0	T
refine-sem (2/3/40)	2967	20523		MErr	20 min	21	MErr	28 min	0	T	< 1	0	T

for that phase requires validity of the bisimulation constraint in the triple graph grammar's axiom. While this could also be proven via implication with composed patterns with the invariant checking tool, the constraints' validity in the three examples is trivially obvious.

For each problem, the following verification approaches and configurations where applied:

1. Enforce with SeekSat and ProCon[Pen09], by recoding the problems as specifications $\{\mathcal{F} \wedge \mathcal{H}\}\mathcal{R}\{\mathcal{F} \vee \neg\mathcal{H}\}$ (cf. Section 8.2),
2. 1-invcheck without extensions,
3. 1-invcheck with impl. – implication for composed graph patterns (Section 7.4) – and
4. 1-invcheck with impl. and including preprocessing (prepr.) to remove some forbidden patterns from the original composed forbidden pattern, because the remaining composed forbidden pattern implies the removed patterns (Theorem 7.40).

Experiments were conducted on the system described in Section 9.1. All counterexamples were written to the disk – runtime for invariant checking includes input and output.

Table 9.1 shows the results. For all six phases of the three examples, both the entire problem (marked in gray) and subproblems were verified, to get more fine-grained results. Given the problem's composed forbidden pattern as $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$, subproblems were created by partitioning I into several sets I_u (for an appropriate index set U), thus splitting the analysis of $\mathrm{Seq}_1(\mathcal{R}, \neg\mathcal{F})$ into $|U|$ problems $\mathrm{Seq}_1(\mathcal{R}, \neg\bigwedge_{i\in I_u} F_i)$. For SeekSat and ProCon, this meant verifying $|U|$ specifications $\{\mathcal{F} \wedge \mathcal{H}\}\mathcal{R}\{\bigwedge_{i\in I_u} \neg F_i \vee \neg\mathcal{H}\}$.

Parts of this comparison between invariant checking and SeekSat/ProCon were already shown and discussed in earlier work *[1, 2]*. Here, however, a newer version of the invariant checker is used; furthermore, the example refine (with phases refine-trans and refine-sem) has been added.

With respect to these examples, we can draw several conclusions:

Interpretation of results. As shown in earlier work *[6]* and reiterated in the table, all required invariants can be verified for all six phases – if implication with composed patterns (impl.) is used. Since the invariants also hold trivially in the triple graph grammars' axioms, all three model transformations preserve behavior in an equivalent or refining manner, respectively.

Applicability of invariant checking. For all three examples (and six phases), the invariant checker is able to prove the expected result – again, if the extension of implication for composed forbidden patterns is used. Verification terminates within reasonable time, especially if preprocessing is used. In particular, a time of 17 minutes (refine-trans) is still acceptable because each phase only needs to be verified once in order to prove behavior preservation for all (infinitely many) instances of the respective model transformation. Note that the OutOfMemoryError came from attempting to write large – and spurious – counterexamples with more than twenty thousand negative application conditions to disk after verification had finished. In general, this can be avoided by configuring the tool to skip the output of counterexamples to files. In this case, it could also have been avoided by eliminating false negatives with implication for composed patterns (see below).

Comparison to Enforce. For all phases in their entirety (and for most subproblems), verification with Enforce (SeekSat/ProCon) is infeasible or, at least, requires significantly more time and memory. Of course, the nature of the problems fits very well to the approach of the invariant checker. Enforce's approach of computing weakest liberal preconditions and their comparison with the specified precondition is costly because of the high number of guaranteed (and forbidden) patterns (and the size of the rules and patterns). Conversely, the restricted approach benefits from a high number of small guaranteed patterns, because many potential counterexamples can be discarded quickly. Furthermore, preprocessing of composed forbidden patterns, which provides an important speed-up (see below), was originally developped particularly because of its application in these examples. Given the existence of the required algorithms in Enforce, implementation of a similar feature may be feasible for Enforce, too.

Finally, it should be noted that the (equivalent) simplification of some negative application conditions for the equiv and refine examples *[6]* would not have been necessary for Enforce because of its support for graph constraints of arbitrary nesting depth. This also applies for several examples where Enforce is well-suited to solve relevant verification problems [Pen09]. One of these examples is a car platooning system [Pen09], which requires additional constraints and modeling effort to create an equivalent specification conforming to the restricted formal model *[1]*.

Other tools and approaches. An earlier version of the invariant checking tool [Dyc12] lacked certain algorithmic optimizations and was therefore unable to verify the systems in acceptable time; an earlier comparison *[1]* noted that a verification attempt was aborted after three days without a result. Even earlier incarnations [BBG^{+}06, BG08b] lacked support for the type of graph constraints and negative application conditions required. Besides Enforce and inductive invariant checking, other verification approaches discussed in Chapter 8 could not be applied in an automated fashion: they either lack an automated implementation, do not support negative application conditions or the type of rules required, or require an initial state or a finite set of initial states. That said, GROOVE could be applied to test singular model transformation instances (but not the transformation as a whole) for bisimilarity. Also, explicit-state model checking approaches in general could be used to generate model transformation instances, e.g. for testing purposes, via the triple graph grammar.

Impact of implication. As explained in Section 6.8 and later in Sections 7.4 and 7.6, the restricted approach to invariant checking is incomplete: it can produce false negatives, particularly because the restricted approach only compares individual patterns by implication. Implication with composed patterns as described in Section 7.4 may reduce this number of false negatives at the cost of performance. In all six problems – and hence, in several subproblems – invariant checking without this extension does indeed lead to false negatives. In some cases (equiv-s-trans), manual inspection may be viable. For equiv-sem, where 59 counterexamples may have thousands of negative application conditions, this does not seem realistic. Using implication with composed patterns (without preprocessing) may require more time, such as the additional ten minutes in equiv-sem. However, all false negatives can be discarded.

Impact of preprocessing. Application of preprocessing – attempting to show implication of each forbidden pattern by the remaining forbidden patterns and the composed guaranteed pattern (Theorem 7.40 (p. 225)) – about halves the time for phases equiv-trans and refine-trans and reduces verification time from nearly half an hour to under a second for equiv-sem and refine-sem. Here, the approach benefits from the structure of the constraints: the largest fragments of the bisimulation constraint are implied by the composed guaranteed pattern and the remaining fragments of the bisimulation constraint – and it takes less than a second to show implication. Then, we can skip the verification (i.e. creation and analysis of s/t-pattern sequences) for those implied constraint fragments, reducing the overall computation time. From the examples discussed here, it appears that there is no reason not to apply preprocessing (in the sense used here). However, given differently structured systems, it may lead to costly and ultimately unsuccessful attempts at proving implication, resulting in an overall performance loss.

Impact of cy^+ and cy^-. It is difficult to judge the exact effect of cy^+ and cy^- if preprocessing is used as this may discard the patterns primarily responsible for large values in both metrics. Without preprocessing, it appears that the number of potential negative application conditions to be created is more important for determining computational effort than the potential number of s/t-pattern sequences – for these examples, at least. This can be seen by directly comparing equiv-trans-msg-mt and equiv-trans-com-mt: the former has a high value of cy^+, a lower value of cy^-, and verification requires 91 seconds; the latter has reverse values of cy^+ and cy^- and requires only 31 seconds. A similar observation can be made for equiv-sem-send and equiv-sem-fire (again, without preprocessing). For Enforce, equiv-s-trans and its subproblems hint at a similar interpretation; in general, the differences in approach of invariant checking and Enforce make it impractical to judge the relationship between computational effort in Enforce and cy^+ or cy^-.

Impact of partial negative application conditions. For the examples used here, partial negative application conditions are not required to improve performance. Rules in the model transformation phases (equiv-trans and refine-trans) create a lot of nodes and connections to existing nodes. Hence, the corresponding reduced rules are isomorphic (or, at least, nearly as large) to the original rules, eliminating the advantage of partial negative application conditions. For the phases concerned with model semantics (equiv-sem and refine-sem), verification is already fast enough, provided preprocessing is used.

Table 9.2. – Behavior preservation and partial negative application conditions

			1-invcheck											
			-			Partial			Impl.,			Impl., partial		
Example ($\vert\mathcal{R}\vert/ \vert\mathcal{F}\vert/ \vert\mathcal{H}\vert$)	cy^+	cy^-	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result
equiv-sem-rt (2/2/44)	14	14	< 1	2	FN	< 1	2	FN	< 1	0	T	< 1	0	T
equiv-sem-init/init (2/2/44)	38	38	< 1	2	FN	< 1	2	FN	< 1	0	T	< 1	0	T
equiv-sem-send (2/1/44)	2941	20497	16 min	19	MErr	8	19	FN	26 min	0	T	18 min	0	T
equiv-sem-fire (2/1/44)	20497	2941	199	36	FN	58	36	FN	5 min	0	T	265	0	T
equiv-sem (2/6/44)	23490	23490	18 min	59	MErr	66	59	FN	29 min	0	T	25 min	0	T
refine-sem-rt (2/1/40)	7	7	< 1	1	FN	< 1	1	FN	< 1	0	T	< 1	0	T
refine-sem-init (2/1/40)	19	19	< 1	1	FN	< 1	1	FN	< 1	0	T	< 1	0	T
refine-sem-send (2/1/40)	2941	20497	20 min	19	MErr	8	19	FN	28 min	0	T	20 min	0	T
refine-sem (2/3/40)	2967	20523	20 min	21	MErr	8	21	FN	28 min	0	T	20 min	0	T

Without preprocessing, however, we can observe interesting effects. In this case, the verification combines semantics rules where only two edges are deleted and created in a large, otherwise unchanging graph with forbidden patterns that contain very similar elements. Table 9.2 shows the application of invcheck with and without implication with composed patterns (impl.) and with and without partial negative application conditions (partial) for equiv-sem and refine-sem. For equiv-sem, the required time for invariant checking without other extensions can be reduced from about twenty minutes to just one minute. Furthermore, since one partial negative application conditions may represent multiple (total) negative application conditions, counterexamples require much less disk space, thusly avoiding the OutOfMemoryError. If implication with composed patterns is used, the effect is not as pronounced: before applying implication, partial negative application conditions are expanded. Still, we save about four minutes. For refine-sem, this value rises to eight minutes.

Although preprocessing makes the use of partial negative application conditions in these examples unnecessary, the comparison in Table 9.2 shows that we can think of systems where they have an important effect on performance.

9.3. Case Study: Shuttle Protocol for Switches

In this section, we will discuss the application of the restricted approach and its extensions to a shuttle protocol focused on handling speed modes when passing over switches. The idea of a protocol for autonomous shuttle is borrowed from the Railcab project, which has served as an example in different versions in the literature before [BBG⁺06, BG08b, SW11].

With the help of the shuttle example, we will discuss the following four aspects: application of k-inductive invariant checking and $k{-}1$-bounded backward model checking to this thesis's running example (Section 9.3.1), a comparison of 1-induction, k-induction, and verification with preconditions and postconditions (Sections 9.3.2 and 9.3.3), the effect of partial negative application conditions for a different version of the shuttle protocol (Section 9.3.4), and verification of systems with attribute constraints and symbolic attributed graph transformation rules (Section 9.3.5).

9.3.1. Verification of Running Example

First, we apply k-inductive invariant checking and $k{-}1$-bounded backward model checking to the running example of this thesis. The protocol for shuttles on switches was used as part of an evaluation of k-inductive invariant checking in earlier work *[3, 4]*. Here, we have extended the safey properties to also prevent shuttles from driving in speed modes acc or brake on a switch, which only leaves slow.

We will discuss the following examples:

shuttle-unsafe: This is our running example (Example 6.1 (p. 111)) with rules s2s, b2s, a2b, and f2b and the alternative rules s2a′, a2f′, and f2f′ (instead of s2a, a2f, and f2f). These rules do not have safeguards in the form of negative application conditions. The composed forbidden pattern forbids shuttles driving in speed modes fast, acc, or brake on a switch. The composed start configuration pattern forbids start graphs with shuttles on switches or in speed modes acc or fast. All system elements are listed in Section C.1.1 of Appendix C.

shuttle-safe: This is our running example (Example 6.1, p. 111) with the regular rules s2s, b2s, a2b, f2b, s2a, a2f, and f2f. The last three rules do have negative application conditions. As before, the composed forbidden pattern forbids shuttles driving in speed modes fast, acc, or brake on a switch. Similarly, the composed start configuration pattern forbids start graphs with shuttles on switches or in speed modes acc or fast. All system elements are listed in Section C.1.2 of Appendix C.

shuttle-single-fault-unsafe: This is the single fault example introduced to demonstrate the effect of forward propagation in Section 7.1, Example 7.1 (p. 170). Sensor faults may lead to a shuttle failing to check for switches ahead; however, we assume that only one sensor fault can occur. The composed forbidden pattern only prevents shuttles from driving fast on a switch. Here, one of the regular rules (applied in the absence of a sensor fault) is missing a negative application condition, leading to an unsafe system. All system elements are listed in Section C.1.3 of Appendix C.

shuttle-single-fault-safe: This is the single fault example introduced to demonstrate the effect of forward propagation in Section 7.1, Example 7.1 (p. 170). As before, sensor faults may lead to a shuttle failing to check for switches ahead; however, we assume that only one sensor fault can occur. The composed forbidden pattern only prevents shuttles from driving fast on a switch. All system elements are listed in Section C.1.4 of Appendix C.

Of course, we usually do not know beforehand whether systems are safe or unsafe – here, we have already seen results in examples in earlier chapters. Also, the unsafe systems were intentionally designed flawed in order to demonstrate verification results for safe and unsafe systems.

First, we will apply k-inductive invariant checking with $k-1$-bounded backward model checking to follow later. For the former, the following configurations were used, all except for the last without output of counterexamples to files (nf.):

- k-invcheck for $k = 1..6$ without extensions,
- k-invcheck for $k = 1..6$ with forward propagation (prop.) for non-negative elements only,
- k-invcheck for $k = 1..6$ with forward propagation including negative application conditions (prop.+NACs), and
- k-invcheck for $k = 1..6$ with forward propagation including negative application conditions and the stop option, i.e. aborting after encountering a counterexample and failing to discard it.

The results are shown in Table 9.3. Here, output of results is not contained in the time values. Again, we can make a number of observations:

Interpretation of results. As expected, the respective composed forbidden patterns are 2-inductive invariants under the corresponding composed guaranteed patterns for shuttle-safe and shuttle-single-fault-safe. Necessarily, they are also k-inductive invariant for any value of $k \geq 2$. The systems shuttle-unsafe and shuttle-single-fault-unsafe do not have the properties as k-inductive invariants for any value of k between 1 and 6.

Table 9.3. – Verification problems and results for shuttle protocol

Example				*k*-invcheck											
				Nf.			Nf., prop.			Nf., prop.+NACs			Prop.+NACs, stop		
$(\lvert\mathcal{R}\rvert / \lvert\mathcal{F}\rvert / \lvert\mathcal{H}\rvert)$	cy^+	cy^-	k	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result
shuttle-unsafe (7/3/15)	546	0	1	< 1	15	F	< 1	15	F	< 1	15	F	< 1	> 0	F
			2	< 1	24	F	< 1	24	F	< 1	24	F	< 1	> 0	F
			3	1	113	F	1	113	F	1	113	F	< 1	> 0	F
			4	6	474	F	6	474	F	6	474	F	< 1	> 0	F
			5	32	2348	F+FN	32	2288	F	32	2288	F	< 1	> 0	F
			6	199	11478	F+FN	196	10384	F	195	10384	F	< 1	> 0	F
shuttle-safe (7/3/15)	546	0	1	< 1	12	F	< 1	12	F	< 1	12	F	< 1	> 0	F
			2	< 1	0	T	< 1	0	T	< 1	0	T	< 1	0	T
			3	< 1	0	T	< 1	0	T	< 1	0	T	< 1	0	T
			4	1	0	T	1	0	T	1	0	T	1	0	T
			5	1	0	T	1	0	T	1	0	T	1	0	T
			6	2	0	T	2	0	T	2	0	T	2	0	T
shuttle-single-fault-unsafe (10/1/16)	260	0	1	< 1	6	F	< 1	6	F	< 1	6	F	< 1	> 0	F
			2	< 1	12	F+FN	< 1	3	F	< 1	3	F	< 1	> 0	F
			3	1	108	F+FN	1	17	F	1	17	F	< 1	> 0	F
			4	9	605	F+FN	2	40	F	2	40	F	< 1	> 0	F
			5	67	3919	F+FN	6	162	F	6	162	F	< 1	> 0	F
			6	10 min	25536	F+FN	28	701	F	30	701	F	1	> 0	F
shuttle-single-fault-safe (10/1/16)	260	0	1	< 1	6	F	< 1	6	F	< 1	6	F	< 1	> 0	F
			2	< 1	9	FN	< 1	0	T	< 1	0	T	< 1	0	T
			3	1	6	FN	< 1	0	T	< 1	0	T	< 1	0	T
			4	6	401	FN	1	0	T	1	0	T	1	0	T
			5	44	2373	FN	1	0	T	1	0	T	1	0	T
			6	6 min	15005	FN	2	0	T	2	0	T	2	0	T

Applicability of k-inductive invariant checking. By k-inductive invariant checking with at least $k = 2$, we can verify the composed forbidden pattern as a 2-inductive invariant for shuttle-safe. For shuttle-single-fault-safe, we additionally require forward propagation. For the unsafe systems, we receive a number of counterexamples in acceptable time. In particular, we can receive counterexamples (if present) for all configurations with the stop option in less than a second for most examples, with two seconds the maximum time required. This is useful if we want to iterate on an unsafely desgined system and repair it to find a k-inductive invariant.

Other approaches and tools. Since the composed forbidden pattern is not a 1-inductive invariant, earlier versions of the tool without k-induction could not have been applied without modifications. In Section 9.3.2, we will discuss to what extent iterative 1-induction could have been applied. Section 9.3.3 will consider whether the problem can be solved by Enforce.

We require negative application conditions for two of the four systems and do not have a singular initial state, which again excludes most other approaches for the problem as specified here. However, given that the state space is finite per initial state, explicit-state model checking could be applied to test the property for singular initial states. Also, we can imagine a graph grammar generating initial states. Then, we could devise a control program that alternates between iterative generation of potential initial states and verification of the safety property with respect to the current initial state (which produces a finite state space).

Still, the number of initial states is infinite; every change of the track topology may lead to an initial state not covered (yet) by the graph grammar. Also, verification per initial state can become rather costly. In a small test topology for shuttle-safe with one shuttle, about 500 tracks connected in a large cycle and two switches in the system, state space exploration with GROOVE took about 15 seconds and created 2006 states with 3506 transitions. It also showed that the system was safe with respect to the specific initial graph.

Impact of k. For systems shuttle-safe and shuttle-single-fault-safe (with prop.) with a 2-inductive invariant, increasing k beyond 2 has little effect; the slight decline in performance for $k \geq 4$ is probably due to additional – and, for these examples, spurious – pipes and filters waiting for input before their termination. However, as mentioned above, a minimum value of $k = 2$ is indeed required to successfully verify the composed forbidden patterns; they are not 1-inductive invariants.

For both unsafe systems, there is a steep increase in the number of counterexample with increasing k – and a similar increase in required time. Unfortunately, due to faults in the systems' designs, increasing k beyond a value of 6 will never lead to successful verification of the desired property as a k-inductive invariant for shuttle-unsafe and shuttle-single-fault-unsafe. Even if we did not know or suspect this, the time required for $k = 6$ (three and ten minutes, respectively) does not raise hopes for verifying $k \geq 7$. Specific values for k will heavily depend on the system in question. Here, it makes the most sense to check individual counterexamples for values of k from 1 to 3, fix potential errors, and rerun the verification.

Impact of forward propagation. Without forward propagation, we cannot successfully (and automatically) verify shuttle-single-fault-safe (for $k \geq 2$). Even for unsafe systems, applying forward propagation will remove a number of false negatives. For shuttle-single-fault-unsafe, this leads to significantly faster verification for $k \geq 4$: while forward propagation requires additional computations, the reduction of false negatives in one step has an impact on the number of potential counterexamples that are created in the next step. Forward propagation of negative application conditions, however, does not have a noticeable effect in these four systems: the

Table 9.4. – Verification problems and results for model checking of shuttle protocol

				k-1-modelcheck											
Example				Nf.			Nf., prop.			Nf., prop.+NACs			Prop.+NACs, stop		
$(\lvert\mathcal{R}\rvert/\lvert\mathcal{F}\rvert/\lvert\mathcal{H}\rvert/\lvert\mathcal{S}\rvert)$	cy^+	cy^-	k	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result
shuttle-unsafe (7/3/15/3)	546	0	2	< 1	3	F	< 1	3	F	< 1	3	F	< 1	> 0	F
			3	< 1	15	F	< 1	15	F	< 1	15	F	< 1	> 0	F
			4	1	71	F	1	71	F	1	71	F	< 1	> 0	F
			5	6	383	F	7	383	F	7	383	F	< 1	> 0	F
			6	37	1945	F	40	1945	F	41	1945	F	< 1	> 0	F
			7	259	10915	F	266	10915	F	274	10915	F	< 1	> 0	F
shuttle-safe (7/3/15/3)	546	0	2	< 1	0	T	< 1	0	T	< 1	0	T	< 1	0	T
			3	< 1	0	T	< 1	0	T	< 1	0	T	< 1	0	T
			4	< 1	0	T	< 1	0	T	< 1	0	T	< 1	0	T
			5	1	0	T	1	0	T	1	0	T	1	0	T
			6	1	0	T	1	0	T	1	0	T	2	0	T
			7	1	0	T	1	0	T	2	0	T	2	0	T
shuttle-single-fault-unsafe (10/1/16/3)	260	0	2	< 1	0	T	< 1	0	T	< 1	0	T	< 1	0	T
			3	< 1	3	FN	< 1	0	T	< 1	0	T	< 1	0	T
			4	1	31	FN	< 1	0	T	1	0	T	< 1	0	T
			5	9	311	F+FN	2	10	F	2	10	F	< 1	> 0	F
			6	75	2207	F+FN	6	91	F	6	91	F	< 1	> 0	F
			7	11 min	14620	F+FN	28	406	F	30	406	F	< 1	> 0	F
shuttle-single-fault-safe (10/1/16/3)	260	0	2	< 1	0	T	< 1	0	T	< 1	0	T	< 1	0	T
			3	< 1	3	T	< 1	0	T	< 1	0	T	< 1	0	T
			4	1	31	FN	1	0	T	< 1	0	T	< 1	0	T
			5	6	243	FN	1	0	T	1	0	T	1	0	T
			6	47	1456	FN	1	0	T	1	0	T	1	0	T
			7	6 min	8914	FN	1	0	T	2	0	T	2	0	T

impact on performance is negligible and, at least here, there are no false negatives to discard because of propagated negative application conditions.

If we want to verify a composed forbidden pattern as an operational invariant, we also need to establish its validity in the k−1-bounded state spaces of all induced graph grammars. In order to do that, we apply Theorem T.3r (p. 149) (or its variants with extensions) and its implementation. In particular, the following configurations were used for shuttle-unsafe, shuttle-safe, shuttle-single-fault-unsafe, and shuttle-single-fault-safe:

- k−1-modelcheck for $k = 2..7$ without extensions,
- k−1-modelcheck for $k = 2..7$ with forward propagation (prop.) for non-negative elements only,
- k−1-modelcheck for $k = 2..7$ with forward propagation including negative application conditions (prop.+NACs), and
- k−1-modelcheck for $k = 2..7$ with forward propagation including negative application conditions and the stop option, i.e. aborting after encountering a counterexample and failing to discard it.

Except for the last, each configuration was used without output of counterexamples to files (nf.). Results are shown in Table 9.4 and lead to the following observations:

Interpretation of results. For shuttle-unsafe, the composed forbidden pattern is not valid even in the 1-bounded state spaces of induced graph grammars. Even if it were a k-inductive invariant, it would not be an operational invariant: there are graph grammars where we can find a violation within one step of the start graph. Results for $k \geq 2$ serve to illustrate performance and the existence of other violations in state spaces with larger bounds. Since violations in a 1-bounded state space are also violations in a 2-bounded state space, each set of counterexamples contains all counterexamples for lower values of k, too.

In the case of shuttle-safe, we can fortunately establish validity of $\mathcal{F}$ – which has been shown to be a 2-inductive invariant – in all 1-bounded state spaces (under the composed guaranteed pattern). Thus, it is an operational invariant. As a result, $\mathcal{F}$ is satisfied in all state spaces (under $\mathcal{H}$) of induced graph grammars. This makes bounded backward model checking for $k \geq 2$ unnecessary; it is shown here for the sake of completeness only.

The system shuttle-single-fault-unsafe exhibits an interesting property: $\mathcal{F}$ is valid in the k-bounded state spaces for $k \leq 3$. However, since it is not a k-inductive invariant, we cannot assume it is an operational invariant. Indeed, bounded backward model checking for $k \geq 4$ yields s/t-pattern sequences that represent transformation sequences from possible start graphs to a violation. Also note that verification without forward propagation leads to a number of false negatives.

Finally, we can also establish the corresponding composed forbidden pattern's validity in the 1-bounded state spaces for shuttle-single-fault-safe; hence, it is an operational invariant by inductive argument. Because of this, any counterexamples for $k \geq 2$ necessarily need to be false negatives, which is confirmed by application of k–1-modelcheck with forward propagation.

Applicability of k–1-bounded backward model checking. Given the results just discussed, we can successfully apply k–1-bounded backward model checking (possibly with extensions) to verify the base case of our inductive argument. We also get symbolic counterexamples for unsafe systems. For lower values of k, we get results with less than a second, implying the approach's feasibility for the examples shown here.

Leaving aside verification via inductive invariants, the approach can also be used to create symbolic traces from possible start graphs of induced graph grammars to violations of $\mathcal{F}$. Those traces then represent both a number (potentially infinite) of transformation sequences and a number (again, possibly infinite) of concrete graph grammars (i.e. graph grammars with a distinguished start graph instead of a symbolic representation by the composed start configuration pattern). However, computational effort increases quickly with higher values of k. Hence, if we require counterexamples for all possible error traces, the approach is likely impracticable for higher values of k. If a single counterexample suffices, the stop option can be used; here, it reduces verification time to under a second.

Other approaches and tools. K–1-bounded backward model checking as described and implemented in this thesis is primarily focused on establishing the base case of our inductive argument. For this purpose and with respect to the shuttle protocol, Enforce might also be applicable (cf. Section 8.2); we discuss this (but with respect to k-inductive invariant checking only) in Section 9.3.3.

K–1-bounded backward model checking as described here is not specifically tailored towards efficiency for other purposes. Suppose we already know about the existence of errors in induced graph grammars' state spaces or suspect it because inductive invariant checking failed to establish a k-kinductive invariant. If we would like to generate and explore concrete error traces from a start graph, explicit model checkers (GROOVE, Henshin) may be better suited. This is particularly true if it is reasonable to assume that most arbitrary start graphs will have violations in their state spaces – and if length of the trace should exceed a certain value (here, for example, $k \geq 7$).

Table 9.5. – Shuttle protocol and iterative 1-induction

					1-invcheck			Enforce		k-invcheck		
					-			(SeekSat/ProCon)		Prop.		
Example ($\vert\mathcal{R}\vert$/ $\vert\mathcal{F}\vert$/ $\vert\mathcal{H}\vert$)	cy^+	cy^-	i	k	time (s)	# ce	result	time (s)	result	time (s)	# ce	result
shuttle-safe (7/3/15)	546	0	1	1	< 1	12	F	1	F			
shuttle-safe-it-2 (7/15/15)	3178	658	2	1	< 1	0	T	4	T			
shuttle-safe (7/3/15)	546	0		2						< 1	0	T
shuttle-single-fault-safe (10/1/16)	260	0	1	1	< 1	6	F	1	F			
shuttle-single-fault-safe-it-2 (10/7/16)	2140	0	2	1	< 1	9	F	3	F			
shuttle-single-fault-safe-it-3 (10/16/16)	6520	0	3	1	1	58	F	7	F			
shuttle-single-fault-safe (10/1/16)	260	0		2						< 1	0	T

Impact of forward propagation. Here, forward propagation does not have to be applied: for both shuttle-safe and shuttle-single-fault-safe, $\mathcal{F}$ is shown to be valid in the 1-bounded state spaces even without forward propagation. Still, it serves to reduce a number of false negatives if we are interested in specific error traces for the unsafe systems. For shuttle-single-fault-safe, forward propagation confirms that counterexamples for $k \geq 2$ are indeed false negatives. We knew that because the composed forbidden property was established as an operational invariant by its nature as a 2-inductive invariant (inductive step) and its validity in the 1-bounded state space (base case).

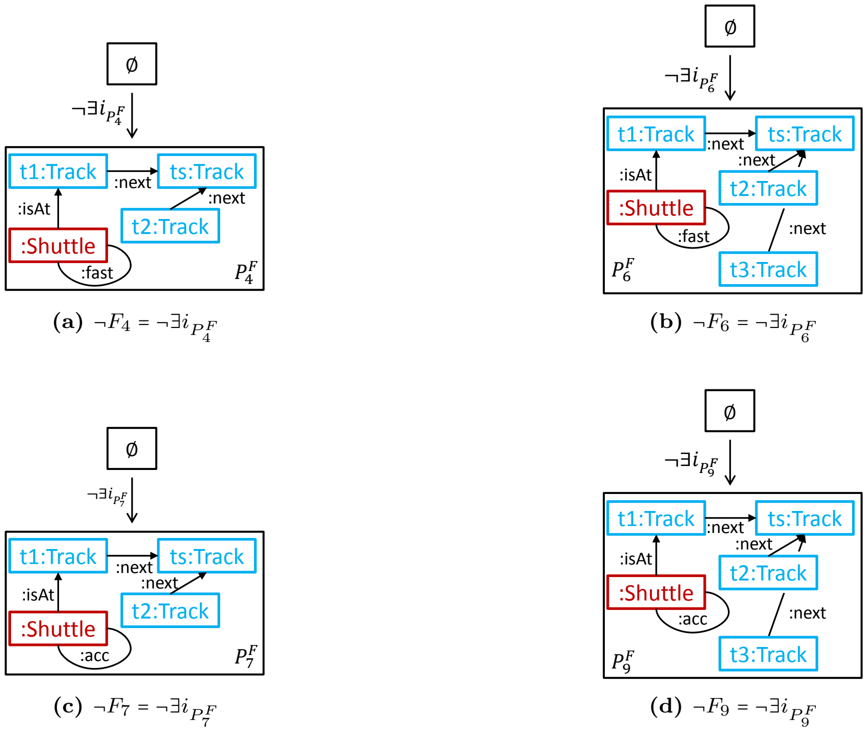
9.3.2. Iterative 1-Induction

In the following, we will iteratively apply 1-inductive invariant checking to gain a result similar to k-inductive invariant checking. Two possible strategies have been described in Section 8.1. Here, we will consider the first strategy and apply it to systems shuttle-safe and shuttle-single-fault-safe from before. We know already that the respective composed forbidden patterns are 2-inductive invariants (under the composed guaranteed pattern).

We will attempt to iteratively enhance the inductive invariant with the goal of establishing a conjunction of the composed forbidden pattern and additional patterns as a 1-inductive invariants. Specifically, we perform 1-inductive invariant checking (1-invcheck), then add any source patterns (their reductions to a graph pattern, that is) in counterexamples to the composed forbidden pattern $\mathcal{F}$, creating a composed forbidden pattern $\mathcal{F}_1$ for the next iteration. This procedure is repeated until verification succeeds – or until we abort the procedure; in general, this approach does not necessarily terminate. Note that the extension of the composed forbidden pattern and the iteration have not been automated, although this is certainly possible. All individual verification tasks in the iteration were also converted to and verified by Enforce; however, each iteration step was created based on counterexamples from the invariant checking tool.

The results are shown in Table 9.5 with the value of i denoting the number of the iteration. For comparison purposes, the results for k-inductive invariant checking with $k = 2$ are shown again.

We have already found out that the respective composed forbidden patterns are not 1-inductive invariant for any system. In the case of shuttle-safe, we can extend the original composed forbidden pattern $\mathcal{F}$ by the twelve counterexamples in the first iteration ($i = 1$). The result – $\mathcal{F}_1$ – can be verified as a 1-inductive invariant, both by invcheck and Enforce. Besides the original forbidden patterns, it contains the four (negated) patterns shown in Figure 9.1, which lead to violations after application of a2b and 2fb, respectively. Four more patterns (not depicted) are the (reduced) source patterns before application of a2f and f2f; as such, they are isomorphic to the former four except for their non-trivial composed negative application



(a) $\neg F_4 = \neg\exists i_{P_4^F}$

(b) $\neg F_6 = \neg\exists i_{P_6^F}$

(c) $\neg F_7 = \neg\exists i_{P_7^F}$

(d) $\neg F_9 = \neg\exists i_{P_9^F}$

Figure 9.1. – Additional forbidden patterns in $\mathcal{F}_1 = \mathcal{F} \wedge \bigwedge_{4 \leq i \leq 15} \neg F_i$

conditions. Those conditions also lead to a non-zero value of cy^- for the second iteration ($i = 2$). The remaining four counterexamples from the first iteration result in patterns isomorphic to patterns already added; in an automated version, checking isomorphisms would be useful to reduce computational effort.

Fortunately, in this case, the additional forbidden patterns are meaningful: as a potential predecessor of a violation, forbidding shuttles in speed mode fast or acc one trask ahead of a switch makes sense. Furthermore, all individual forbidden patterns are still implied by the composed start configuration pattern, which forbids shuttles in speed modes acc or fast (and shuttles on switches). For this system, with respect to the operational invariance of the targeted composed forbidden pattern, we could get similar results for both 2-induction and iterative 1-induction (altough the latter would require automation of the iteration to be truly comparable).

The situation is different for shuttle-single-fault-safe. As documented in Table 9.5, the accumulated composed forbidden patterns cannot be verified as 1-inductive invariants for $i = 1..3$ – and, although not shown for here, not for any further value of i. The restricted approach does not include guaranteed patterns in the sequences it creates – it relies on analyzing the resulting source and target patterns to take earlier violations of forbidden or guaranteed patterns into account. It requires forward propagation to eliminate potential false negatives that are caused by this strategy. In the iterative application of 1-inductive invariant checking, this aspect is missing: information accessible by forward propagation is not available here.

Of course, we could change the creation of s/t-pattern sequences to incorporate guaranteed patterns and other forbidden patterns in source and target patterns, which would then appear in the forbidden pattern accumulated by each iteration. In essence, this is what the general approach (Chapter 5) does – and iteration of the general approach for $k = 1$ could indeed lead to the desired result here and allow verification of an extended composed forbidden pattern as a 1-inductive invariant. However, we have had good reason to choose the restricted and not

Table 9.6. – Comparison of k-invcheck and Enforce for shuttle protocol

				k-invcheck			Enforce	
				Nf., prop.+NACs			(SeekSat/ProCon)	
Example ($\lvert\mathcal{R}\rvert$ / $\lvert\mathcal{F}\rvert$ / $\lvert\mathcal{H}\rvert$)	cy^+	cy^-	k	time (s)	# ce	result	time (s)	result
shuttle-unsafe (7/3/15) (Enforce: 8/3/15)	546	0	1	< 1	15	F	1	F
			2	< 1	24	F	15	F
			3	1	113	F	82	F
			4	6	474	F	7 min	F
shuttle-safe (7/3/15) (Enforce: 8/3/15)	546	0	1	< 1	12	F	1	F
			2	< 1	0	T	24	T
			3	< 1	0	T	289	T
			4	1	0	T	39 min	T
shuttle-single-fault-unsafe (10/1/16) (Enforce: 11/1/16)	260	0	1	< 1	6	F	1	F
			2	< 1	3	F	17	F
			3	1	17	F	228	F
			4	2	40	F	43 min	F
shuttle-single-fault-safe (10/1/16) (Enforce: 11/1/16)	260	0	1	< 1	6	F	1	F
			2	< 1	0	T	18	T
			3	< 1	0	T	242	T
			4	1	0	T	35 min	T

general approach for implementation – mainly its computational effort and visual complexity.

There is another detrimental effect of lacking forward propagation in iteration of 1-induction with the restricted approach (for shuttle-single-fault-safe). For $i = 3$, we get forbidden patterns describing a shuttle in speed mode slow two tracks ahead of a switch. Obviously, this describes perfectly legitimate states that should not be forbidden just because two applications of rules with sensor faults (s2a′ and a2f′) lead to a violation.

In summary, iterative application of the restricted approach to 1-inductive invariant checking cannot, in general, simulate k-induction, although it may work for some systems.

9.3.3. Enforce and k-Induction

In Sections 8.1 and 8.2, we have discussed two techniques to employ Enforce and recode the respective system specification to approximate or emulate k-inductive invariant checking. Here, we will discuss the second technique (Section 8.2), because (with the exception of adapting the specifications) it can be executed automatically with the tool and implementation available.

Enforce is able to compute weakest liberal preconditions for rules and graph programs but does not usually analyze intermediate results. Hence, verification of a specification $\{\mathcal{F} \wedge \mathcal{H}\}\{\mathcal{R};\mathcal{R}\}\{\mathcal{F} \vee \neg\mathcal{H}\}$ investigates all sequences of applications of two rules from $\mathcal{R}$ and checks preconditions and postconditions – but it does not check the result in between for forbidden or guaranteed patterns. We have seen that this analysis can be taken care of by adding a rule addC that requires validity of $\mathcal{F}$ and $\mathcal{H}$ to be applicable. By changing the program to $\{\mathcal{F} \wedge \mathcal{H}\}\{\mathcal{R};\mathsf{addC};\mathcal{R}\}\{\mathcal{F} \vee \neg\mathcal{H}\}$, we get the required result for 2-induction. This procedure can be extended for arbitrary values of k.

Table 9.6 shows the results for the application of this procedure to shuttle-unsafe, shuttle-safe, shuttle-single-fault-unsafe, and shuttle-single-fault-safe. Results for k-invcheck with forward propagation are listed here again for comparison purposes. The procedure yields the same general results (with the exception of specific counterexamples) for all systems, although Enforce requires more time and, for three of the four systems, will likely exceed the time limit of one hour for $k = 5$. Also, Enforce requires more time for increasing values of k even if the property is a k-inductive invariant for a lower value. This means that when applying Enforce in this fashion to find a k-inductive invariant without an idea of the value of k, one should always start with a low value of k and gradually increase it until verification succeeds (or until we decide to abort the process). This is not required for the invariant checker if the respective property can be shown as a k-inductive invariant for some realistic value of k. As soon as the combination of the Seq-construction and the analysis of s/t-pattern sequences reaches that value of k, all counterexamples are discarded and are not extended for higher values of k specified in the configuration.

In conclusion, performing k-induction with Enforce is possible, but, for the problems discussed here, seems impractical by comparison, especially given values of k above 2. This is reinforced by the results of Section 9.2 for systems with a high number of forbidden and guaranteed properties. It seems likely that this is due to the necessity of shifting all forbidden and all guaranteed patterns over the graph program (here: application of graph rules). Here, this problem gets worse because the addC rule also needs to do this in between regular rule applications. Again, Enforce was not designed to perform these tasks, let alone optimized for it. Its strengths lie elsewhere, such as in its support for graph programs and application conditions and graph constraints of arbitrary nesting depth, which do not conform to the restricted formal model discussed in this thesis. Successful application of Enforce to meaningful program specification has been discussed in the literature [Pen08a, Pen08b, Pen09].

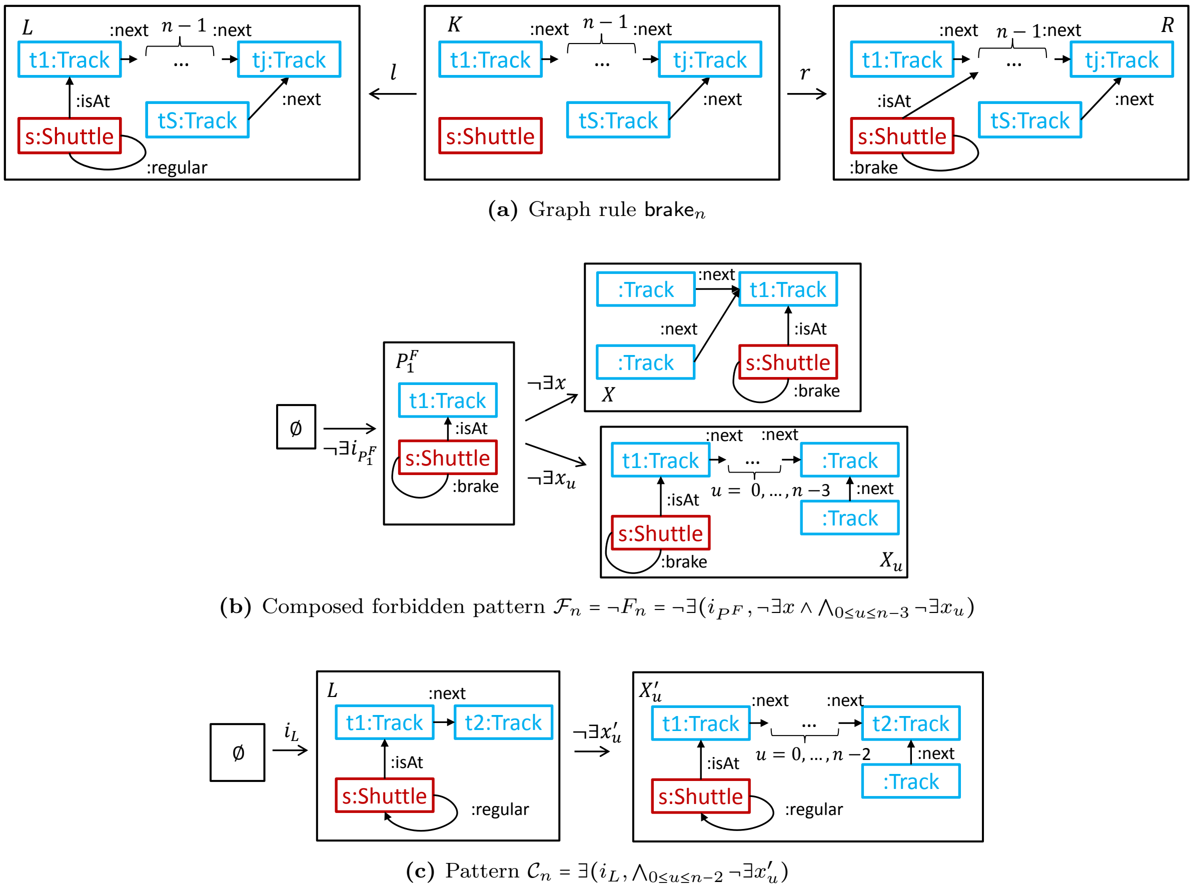
9.3.4. Partial Negative Application Conditions

The following example picks up the running example used in Section 7.3, Example 7.15 (p. 189). It demonstrates the impact of partial negative application conditions in a scenario constructed specifically for this purposes. The example is only a fragment of a potential system; as such, it is both minimal and synthetic. However, this fragment or fragments with similar features could arguably be found as part of a complete system, too.

Example 9.1 (system fragments shuttle-brake-late-n)**.** In these system fragments, we have a graph rule brake_n that causes a shuttle to switch from a regular driving mode to mode brake if there is a switch n tracks ahead. The rule is depicted in Figure 9.2(a) with an abbreviated notation for the respective number of tracks between the shuttle's current track and a switch. We analyze values for j from 2 to 8; each value induces a new system fragment shuttle-brake-late-n with the respective single rule $\mathcal{R} = \{\mathsf{brake}_n\}$.

For each system fragment, we have a composed forbidden pattern $\mathcal{F}_n = \neg F_n = \neg\exists(i_{PF}, ac_n)$, whose composed negative application condition $ac_n = \neg\exists x \wedge \bigwedge_{0\leq u\leq n-3} \neg\exists x_u$ depends on the value of n. The pattern is shown in Figure 9.2(b). It expresses that a shuttle should not be in braking mode unless its current track is a switch or there is a switch between 1 and $n-2$ tracks ahead. As such, the rule brake_n likely causes a shuttle to brake unnecessarily early. By verifying $\mathcal{F}_n$, we can detect this behavior. We also consider a composed guaranteed pattern $\mathcal{H}$; it consists of seven guaranteed patterns that model cardinality constraints and forbid the existence of more than one shuttle.

We also investigate systems shuttle-brake-late-prio-n, which have an additional aspect: a rule $\mathsf{moveRegularly}_n$ of higher priority than brake_n may prevent braking when the shuttle is not too close to a switch. Here, too close means that a switch is at most $n-1$ tracks ahead. For purposes of analyzing counterexamples for rules with priorities (cf. Section 7.2, particularly Theorem T.2e-rp), we compare source patterns of an s/t-pattern sequence with the rule's applicability condition. Here, we only show the applicability condition (that is, its reduction to a pattern) of $\mathsf{modeRegularly}_n$ (Figure 9.2(c)): $C_n = \exists(i_L, ac_{C_n})$ with the composed negative application condition $ac_{C_n} = \bigwedge_{0\leq u\leq n-2} \neg\exists x'_u$ dependent on n. All elements of the system fragment are listed in Section C.1.5 of Appendix C.



(a) Graph rule brake_n

(b) Composed forbidden pattern $\mathcal{F}_n = \neg F_n = \neg\exists(i_{P^F}, \neg\exists x \wedge \bigwedge_{0\leq u\leq n-3} \neg\exists x_u)$

(c) Pattern $\mathcal{C}_n = \exists(i_L, \bigwedge_{0\leq u\leq n-2} \neg\exists x'_u)$

Figure 9.2. – Graph rules brake_n, forbidden patterns F_n, and patterns C_n

Table 9.7. – Fragment of shuttle protocol and partial negative application conditions

				1-invcheck								
				Prio.			Prio., nf.			Prio., partial		
Example ($\lvert\mathcal{R}\rvert$/ $\lvert\mathcal{F}\rvert$/ $\lvert\mathcal{H}\rvert$)	n	cy^+	cy^-	time (s)	# ce	result	time (s)	# ce	result	time (s)	# ce	result
shuttle-brake-late-2 (1/1/7)	2	10	21	< 1	1	F	< 1	1	F	< 1	1	F
shuttle-brake-late-3 (1/1/7)	3	12	62	< 1	1	F	< 1	1	F	< 1	1	F
shuttle-brake-late-4 (1/1/7)	4	14	315	< 1	1	F	< 1	1	F	< 1	1	F
shuttle-brake-late-5 (1/1/7)	5	16	2433	< 1	1	F	< 1	1	F	< 1	1	F
shuttle-brake-late-6 (1/1/7)	6	18	23149	5	1	F	1	1	F	< 1	1	F
shuttle-brake-late-7 (1/1/7)	7	20	250260	144	1	MErr	11	1	F	< 1	1	F
shuttle-brake-late-8 (1/1/7)	8	22	2999187			MErr			MErr	< 1	1	F
shuttle-brake-late-prio-2 (2/1/7)	2	16	28	< 1	0	T	< 1	0	T	< 1	0	T
shuttle-brake-late-prio-3 (2/1/7)	3	18	76	< 1	0	T	< 1	0	T	< 1	0	T
shuttle-brake-late-prio-4 (2/1/7)	4	20	342	< 1	0	T	< 1	0	T	< 1	0	T
shuttle-brake-late-prio-5 (2/1/7)	5	22	2481	7	0	T	7	0	T	< 1	0	T
shuttle-brake-late-prio-6 (2/1/7)	6	24	23228	29 min	0	T	29 min	0	T	< 1	0	T
shuttle-brake-late-prio-7 (2/1/7)	7	26	250381			TmOut			TmOut	< 1	0	T
shuttle-brake-late-prio-8 (2/1/7)	8	28	2999366			MErr			MErr	< 1	0	T

We applied 1-invcheck with and without partial negative application conditions (partial) to the system fragments shuttle-brake-late-n and shuttle-brake-late-prio-n with n ranging from 2 to 7. The results are shown in Table 9.7. We can draw the following conclusions:

Interpretation of results. For system fragments shuttle-brake-late-n, the shuttle starts braking too early, as expected, and we get a corresponding counterexample. In system fragments shuttle-brake-late-n, the rules moveRegularly$_n$ have a higher priority than brake$_n$. They are only applicable if a switch is not too close. Here, they prevent the shuttle from braking too early. Hence, the composed forbidden pattern, which forbids a shuttle in mode braking without a switch closer than $n - 1$ tracks ahead, is indeed a 1-inductive invariant for brake$_n$.

Impact of partial negative application conditions. In this example, the benefit of partial negative application conditions is obvious: we are able to verify all systems in less than a second – and, where the composed forbidden pattern ist not a 1-inductive invariant, we get a counterexample. Without partial negative application conditions, verification cannot be performed for $n = 8$ with the given parameters – and for shuttle-brake-late-7, saving the counterexample to disk causes an OutOfMemoryError. In the case of shuttle-brake-late-prio-n, the additional comparison (by implication) of the counterexample and C_n (Figure 9.2(c)) leads to an even steeper increase of verification times: for $n = 5$, we need only 7 seconds while $n = 6$ requires about 29 minutes. Verifying $n = 7$ would need more than an hour – and $n = 8$ requires more memory.

Of course, the situation was deliberately created to show the effect of partial negative application conditions in an extreme case. The reduced rule of brake$_n$ is significantly smaller than the rules themselves – in particular, the smallest reduced rule of brake$_2$ is also a reduced rule for rules with larger values of n. Thus, even for large values of n, the effort of shifting negative application conditions from the forbidden pattern remains negligible if partial application conditions are used. Likewise, the nature of the pattern C_n (created from the rule moveRegularly) allows a comparison (by implication) with source patterns in counterexamples without expanding partial negative application conditions to the complete context of the source pattern.

It is difficult to generalize the impact of partial negative application conditions from this small example. While the effect is noticeable here, further experiments with complete systems are required to draw any conclusion as to their importance in practice. Furthermore, while the experiments could also be part of a Seq-construction for values of $k \geq 2$, we have not experimented with partial negative application conditions in that context, i.e. for k-inductive invariant checking with $k \geq 2$. The current procedure requires expansion of partial negative application conditions after analyzing the counterexamples (by implication) and before incrementing k; this could diminish the expected effect.

Impact of cy^-. The value of cy^- determines the maximum number of negative application conditions (when only considering nodes) in a target pattern. As such – and given the form of the system fragments – cy^- gives a very clear indication about the computational effort required if all such conditions are considered. A partial negative application condition, on the other hand, may encode a much larger number of (total) negative application conditions, which explains the connection between cy^- and the required time and effort with and without partial negative application conditions.

9.3.5. Attribute Constraints and Symbolic Attributed Rules

The restricted approach to k-inductive invariant checking and k−1-bounded backward model checking requires systems and specifications formalized as typed graph transformation systems and typed graph constraints conforming to the restricted formal model. One obvious idea of extending this approach is to allow the use of attributes and attribute constraints in graph rules and graph constraints. This requires using an adequate formalism for attributes and extending the symbolic encoding, construction of s/t-pattern sequences, and analysis of s/t-pattern sequences to said formalism – and finally, implementing those extensions.

There exists earlier work [Nic16] describing the implementation of support for attributes and attribute constraints for 1-inductive invariant checking. In the context of this thesis, this implementation has been extended in a prototypical fashion to also cover support for attributes and attribute constraints in k-inductive invariant checking. However, contrary to the restricted approach to k-inductive invariant checking discussed in this thesis (i.e. without attributes), it has not been formally described and proven. Thus, we cannot place the same faith in the implementation and verification results. Instead, the primary purpose of this example is to demonstrate the usefulness of support for attributes in k-inductive invariant checking and outline the general idea and feasibility of its implementation.

The use of attributes and attribute constraints in this example follows the notions of *attributed graphs* [OL10b], *symbolic (typed) attributed graphs* [OL10b, SLO18] and *symbolic (typed) attributed graph transformations* [OL10b, OL10a] as described by Schneider, Lambers, and Orejas. Attributed graphs are graphs that include node and edge attributes; each attribute is mapped to its node or edge on one end and an attribute value (of the corresponding sort) on the other end [OL10b].[3] In symbolic attributed graphs, however, this attribution maps each attribute to a node or edge (as before) – and to an attribute variable instead of an attribute value [SLO18, OL10b]. These attribute variables of the symbolic attributed graph are constrained by a term that evaluates to a Boolean value and that is defined over attribute variables and operations (including constants) of the respective sorts [SLO18]. Evaluation of terms and sets of sorts and operations are described by an algebraic specification that is also part of the symbolic attributed graph [SLO18].

As such, a symbolic attributed graph specifies a number of attributed graphs [OL10b]. Intuitively speaking, a symbolic attributed graph represents all attributed graphs that are iden-

[3] Formally, attributed graphs are so-called *E-graphs* [OL10b, EEPT06].

tical with respect to nodes, edges, and attributes, and whose attribute assignments satisfy the symbolic attributed graph's (attribute) constraint [OL10b]. A *grounded symbolic attributed graph* describes exactly one attributed graph by constraining attribute variables appropriately [OL10b, SLO18]. Symbolic attributed graphs can also be typed over a type graph [SLO18].

Nested application conditions and nested graph constraints can be extended over symbolic (typed) attributed graphs [SLO18] (called *graph conditions* and *graph properties* in the original source). As for non-attributed application conditions, satisfiability is defined by graph morphisms; in particular, by symbolic attributed (typed) graph morphisms [SLO18].

A symbolic graph transformation rule is defined as a pair of a graph rule over graphs with attributions (see above) and a constraint [OL10a]. If successfully applied to a grounded symbolic attributed graph, the result is again grounded [OL10b].

In the current implementation and the example below, we use the concepts sketched above as follows: node types (but not edge types) in type graphs may specify attribute types of various sorts; nodes of *(symbolic) attributed graphs* may then have attributes with attribute values (attribute variables) of the corresponding sorts. However, such there may only be one attribute variable or value per node and attribute type.

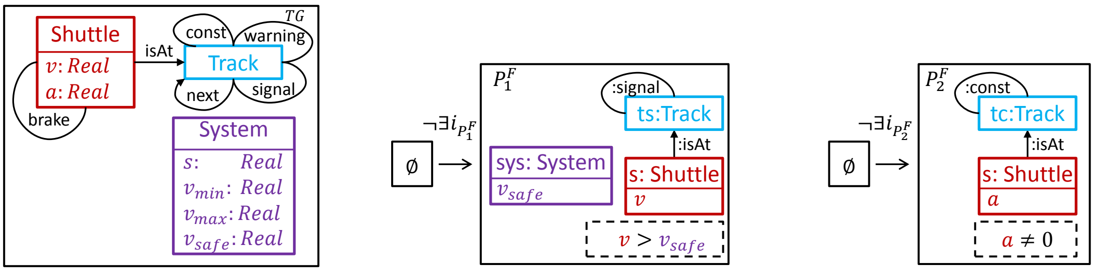
Graph patterns are extended to use symbolic attributed graphs – but only attributes in a pattern's existential condition may be constrained, not attributes in composed negative application conditions. Similarly, graph rules may be constrained (with respect to attributes), but composed negative application conditions may not. For now, these restrictions simply follow the state of the implementation.

Example 9.2 (shuttle protocol with attributes)**.** Figure 9.3(a) shows the extended type graph for systems shuttle-attributes-n. As before, we have connected tracks with shuttles positioned on tracks. In addition, tracks may contain signals and warnings. The idea is that warnings announce a signal a certain number of tracks ahead – and shuttles should not pass a signal above a certain velocity. Tracks can also be marked with const, which indicates that a track should be passed at constant velocity – for safety purposes or reasons of passenger comfort.

Shuttles have a new braking mode. Speed settings, which were modeled as flags (reflexive edges) before, are now specified in two real-valued attributes that denote acceleration (a) and velocity (v). There is an additional node type System that stores system parameters: track length (s), which is fixed for all tracks, minimum and maximum velocity of shuttles (v_{min} and v_{max}), and the velocity that should not be exceeded when passing signals (v_{safe}). There may only be one global system node; this is enforced by a guaranteed constraint (see below).

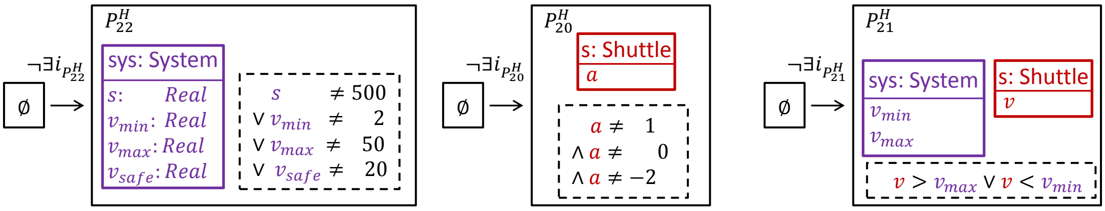
All rules and patterns of the example are shown in Section C.1.6 of Appendix C; here, we will only discuss the most important elements. Figures 9.3(b) and 9.3(c) show two forbidden patterns. First, shuttles should not pass signals above the specified safe velocity v_{safe}. Second, shuttles may not accelerate or decelerate when on tracks that require constant velocity. Formally, both patterns are graph constraints over symbolic attributed graphs: they have an attribute constraints $v > v_{safe}$ and $a \neq 0$, respectively. Hence, an attributed graph with the system element and with a shuttle on a track violates $\neg F_1$ only if the shuttle's velocity exceeds v_{safe}. Formally, v and v_{safe} are attributes with attribute variables attached; then, the constraint is defined over the attribute variables, not the attributes themselves. To improve readability, however, we do not make this distinction in the figures.

We specify (v_{safe}) and the other system parameters in a guaranteed constraint $\neg H_{22}$ (Figure 9.4(a)): minimum and maximum velocity are 2 m/s and 50 m/s, respectively, and the threshold on velocity for passing signals is 20 m/s. Tracks have a uniform length of 500 m. Furthermore, as shown in Figure 9.4(b), shuttles can only accelerate with 1 m/s^2, decelerate with −2 m/s^2 or keep their velocity with an acceleration of 0 m/s^2. If system parameters change – because, for instance, changed regulations require lowering v_{safe} or shuttles should be able



(a) Type graph TG **(b)** Forbidden pattern $\neg F_1$ **(c)** Forbidden pattern $\neg F_2$

Figure 9.3. – Type graph and forbidden patterns



(a) Pattern $\neg H_{22} = \neg\exists i_{P_{22}^H}$ **(b)** Pattern $\neg H_{20} = \neg\exists i_{P_{20}^H}$ **(c)** Pattern $\neg H_{21} = \neg\exists i_{P_{21}^H}$

Figure 9.4. – Guaranteed patterns modeling system parameters

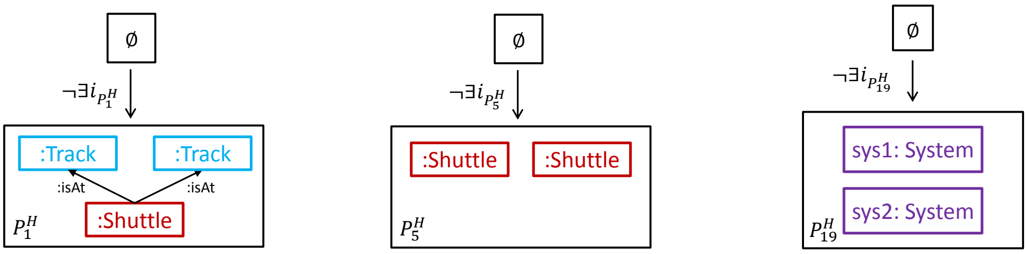
to accelerate faster – we need to update these patterns' attribute constraints.

We also exclude situations (Figure 9.4(c)) where shuttles fall below the minimum velocity or exceed their maximum velocity. Usually, we may want to formulate appropriate rules to ensure this last property and verify it – here, we leave this part out and may, for example, assume that shuttles are built such that the thresholds on velocity cannot be violated.

There are several cardinality constraints, some of which are shown in Figure 9.5. Again, a shuttle cannot be on two tracks at the same time and we verify only one shuttle at a time. Also, there is only a single (global) system element.

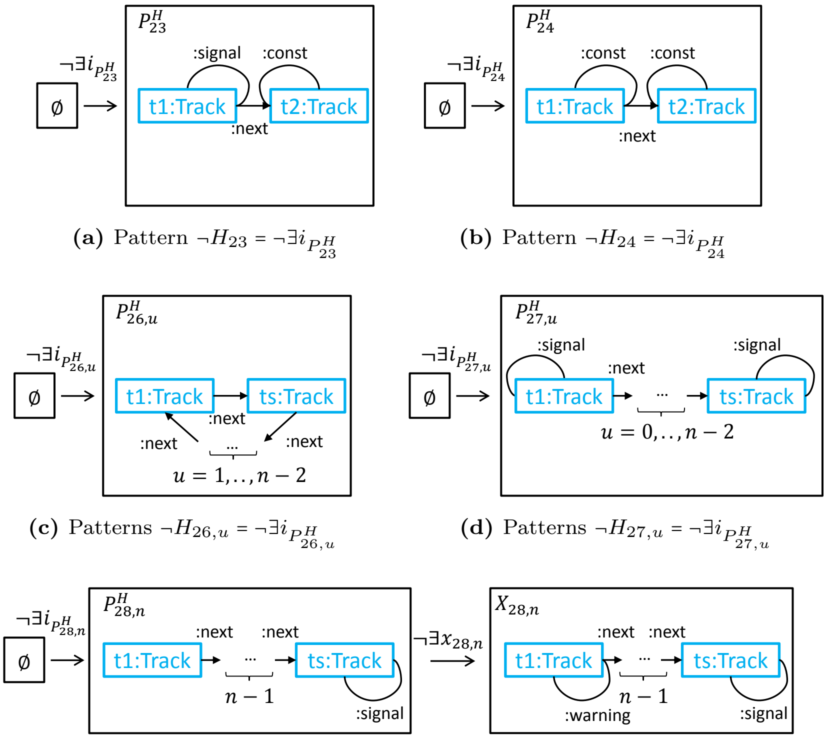
We also specify certain properties of the system's track topology (Figure 9.6): a track that requires constant velocity cannot follow directly after a signal (Figure 9.6(a)) and tracks that require constant velocity cannot follow directly after each other (Figure 9.6(b)). Pattern $\neg H_{26,u}$ (Figure 9.6(c)) is actually a set of patterns that depend on a parameter n (with $1 \leq u \leq n-2$): together, they express that there are no cycles of n or fewer tracks. Similarly, patterns $\neg H_{26,u}$ specify that signals have at least $n-2$ tracks between them. Finally, $\neg H_{28,n}$ requires that each signal has a corresponding warning on each track n tracks ahead of the signal.

The idea of the parameter n is that warnings are placed a certain number of tracks ahead of a signal ($\neg H_{28,n}$). The question then is whether a shuttle can decelerate in time for a certain



(a) Pattern $\neg H_1 = \neg\exists i_{P_1^H}$ **(b)** Pattern $\neg H_5 = \neg\exists i_{P_5^H}$ **(c)** Pattern $\neg H_{19} = \neg\exists i_{P_{19}^H}$

Figure 9.5. – Guaranteed patterns modeling cardinality constraints



(a) Pattern $\neg H_{23} = \neg\exists i_{P^H_{23}}$ **(b)** Pattern $\neg H_{24} = \neg\exists i_{P^H_{24}}$

(c) Patterns $\neg H_{26,u} = \neg\exists i_{P^H_{26,u}}$ **(d)** Patterns $\neg H_{27,u} = \neg\exists i_{P^H_{27,u}}$

(e) Pattern $\neg H_{28,n} = \neg\exists(i_{P^H_{28,n}}, \neg\exists x_{28,n})$

Figure 9.6. – Guaranteed patterns modeling track topology constraints

value of n.

There are nine symbolic graph rules, two of which are depicted in Figure 9.7. Rule toSteady-const-warning (Figure 9.7(a)) is applicable if a shuttle is on a track with a warning and without a signal ($\neg\exists x_1$) and there is a track directly ahead that requires constant velocity. Application of the rule causes the shuttle to enter braking mode and set its acceleration to zero ($a' = 0$). Since the subsequent track requires constant velocity, the shuttle cannot yet decelerate.

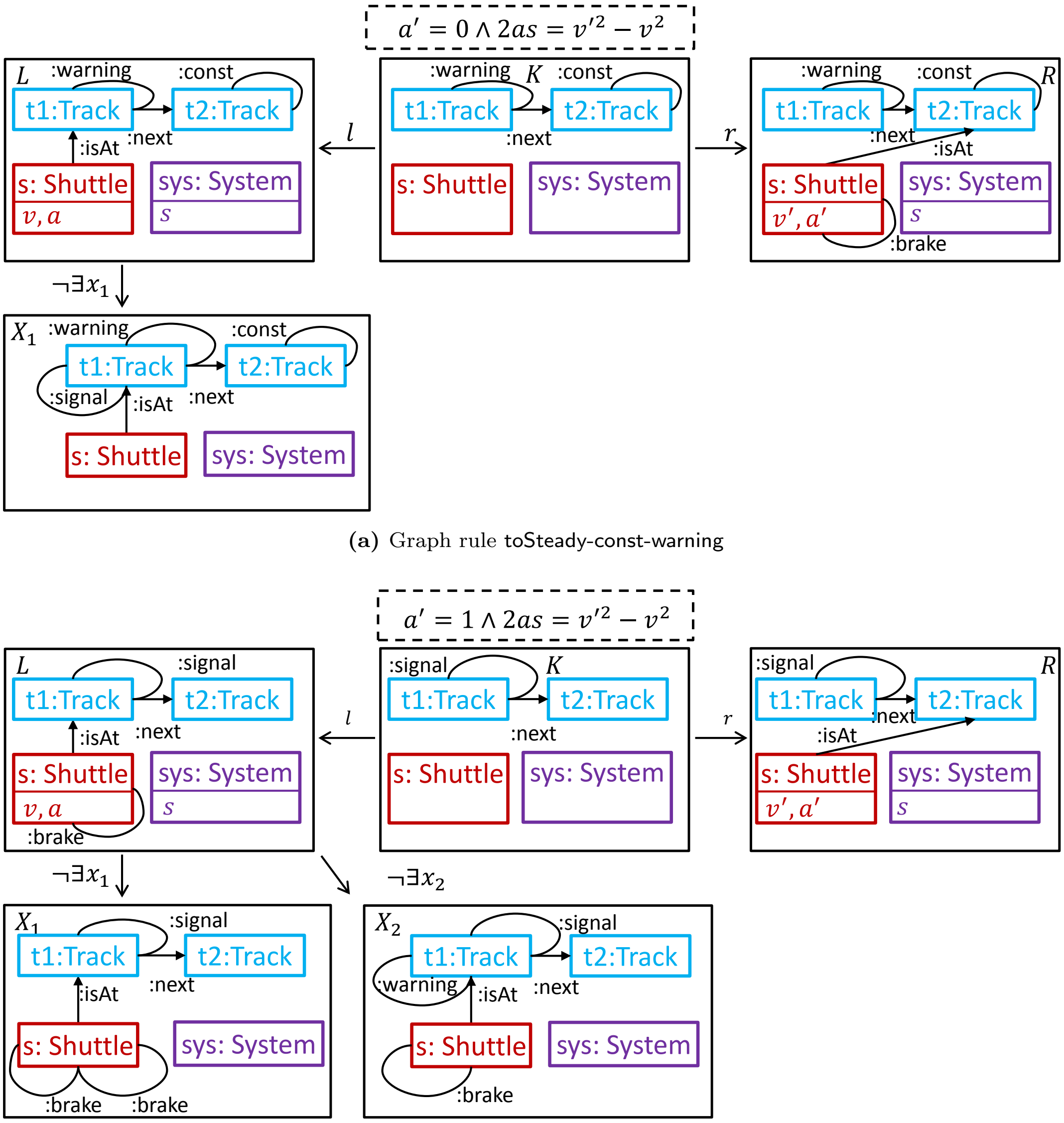
The rule toAcc-remBrake (Figure 9.7(b)) is applicable if a shuttle is in braking mode, has arrived on a track with a signal but without a warning ($\neg\exists x_2$), and does not have a second brake flag set ($\neg\exists x_1$). Then, the shuttle leaves braking mode and sets its acceleration to $1\,\mathrm{m/s}^2$.

In both rules, attributes v and a refer to the attributes on the left side of the rule. Attributes v' and a' refer to the right rule side; s does not change. Formally, the attributes have attribute variables attached and the constraint refers to the variables. As before, we have omitted this part to improve readability.

The constraint $2as = v'^2 - v^2$ is a result of the laws governing movement with uniform acceleration: given acceleration a and a starting velocity of v, velocity v' after time (t) has passed can be computed as $v' = at + v$. Likewise, distance passed is $s = \frac{1}{2}at^2 + vt$. Combining both and eliminating t gives us $2as = v'^2 - v^2$. Note that shuttles only change their acceleration after arriving on a track. Thus, in order to calculate the time (and change in velocity) for passing a track, we still use the value a on the left rule side (as opposed to $2a's = v'^2 - v^2$).

There are several other rules not shown here. All follow similar patterns: acceleration can be set to 0, 1, or −2 and – depending on warnings and signals – shuttles will enter or leave braking mode. All elements are depicted in Section C.1.6 of Appendix C. △

Inductive invariant checking for systems with symbolic graph rules and graph constraints over symbolic attributed graphs is performed in two steps: first, we consider only the plain



(a) Graph rule toSteady-const-warning

(b) Graph rule toAcc-remBrake

Figure 9.7. – Graph rules toSteady-const-warning and toAcc-remBrake

graphs of the elements involved; secondly, we integrate the attributes and attribute constraints.

More specifically, we first create s/t-pattern sequences from forbidden patterns as usual (Seq-construction, Theorem T.1r (p. 130)) and ignore the attribute-related elements in graph constraints and graph rules. We also analyze sequences for violations of other forbidden patterns or guaranteed patterns (Theorem T.2r (p. 143)). This involves checking implication (by Theorem 6.8 (p. 120)) between reduced source and target patterns and forbidden and guaranteed patterns. If patterns in a sequence imply a forbidden or guaranteed pattern without attribute constraints, the sequence can be discarded – otherwise, all morphisms showing implication as described by Theorem 6.8 (p. 120) are saved for later analysis.

The second step analyzes each s/t-pattern sequence by combining all attribute constraints relevant for the sequence into a constraint system. In particular, this involves attribute constraints of the forbidden pattern and graph rules used to create the sequence. It also involves attribute constraints from other forbidden and guaranteed patterns, who could be implied by reduced source and target patterns in the sequence via morphisms. When transferring attribute constraints over graph, variables need to be substituted accordingly.

Finally, the constraint solver Z3 is used to find a solution – an attribute valuation – such that the constraint system is satisfied. Then, a transformation sequence (with plain graphs) that satisfies the s/t-pattern sequence (disregarding attribute constraints), can be combined with the solution to the constraint system to form a sequence of attributed graph transformations such that:

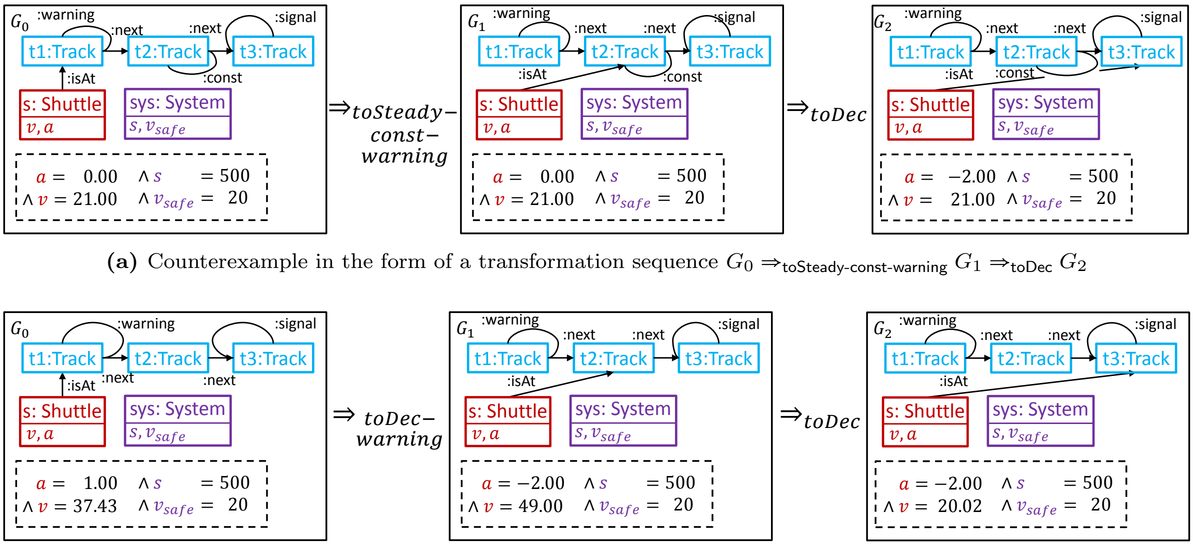
- the rightmost graph violates the forbidden pattern $\neg F$ used to create the s/t-pattern sequence (i.e. satisfies its non-negated form F)
- all symbolic graph rules in the sequence are indeed applicable,
- no graph in the sequence violates a guaranteed pattern, and
- no graph except the rightmost graph violates a forbidden pattern.

In other words, the transformation sequence with the attribute valuation found by the constraint solver is a counterexample for the (composed) forbidden pattern being a k-inductive invariant with respect to sequences of symbolic graph transformations applied to attributed graphs (or grounded graphs).

For now, inductive invariant checking with attributes does not support most of the extensions available for the restricted approach. While rule priorities could be supported without much additional effort, adding forward propagation would be more involved. Implication with composed graph patterns would be the most challenging of the four extensions.

Example 9.3. Verification was performed with n ranging from 2 to 5, i.e. with shuttles being notified about upcoming signals two, three, four, or five tracks ahead ($\neg H_{28,n}$, Figure 9.6(e), p. 263) – and with a corresponding minimum distance between signals ($\neg H_{27,u}$, Figure 9.6(d), p. 263). System parameters were set to the values used in pattern $\neg H_{22}$ (Figure 9.4(a), p. 262). Since shuttles had to pass n tracks between a warning and the corresponding signal (including both the warning and the signal), it made sense to apply k-inductive invariant checking with $k = n$.

The constraint $\neg F_2$ (Figure 9.3(b), p. 262) is a 1-inductive invariant for all systems; composed negative application conditions in graph rules make sure that only rules toSteady... with $a' = 0$ can be applied when a shuttle is about to move to a track that requires constant velocity. The situation is different for $\neg F_1$ (Figure 9.3(b), p. 262). While it is a 5-inductive invariant for $n = 5$ (i.e. shuttle-attributes-5), there are counterexamples for n ranging from 2 to 4. They are shown and discussed below in the form of transformation sequences using attributed graphs. The invariant checker actually provides s/t-pattern sequences with specific attribute valuations, i.e. s/t-pattern sequences with patterns defined over grounded (symbolic attributed) graphs.


G_0
:warning
:next
:next
:signal
t1:Track
t2:Track
t3:Track
:isAt
:const
s: Shuttle
v, a
sys: System
s, v_{safe}
$a = 0.00 \wedge s = 500$
$\wedge v = 21.00 \wedge v_{safe} = 20$
⇒toSteady−const−warning
G_1
$a = 0.00 \wedge s = 500$
$\wedge v = 21.00 \wedge v_{safe} = 20$
⇒toDec
G_2
$a = -2.00 \wedge s = 500$
$\wedge v = 21.00 \wedge v_{safe} = 20$
$a = 1.00 \wedge s = 500$
$\wedge v = 37.43 \wedge v_{safe} = 20$
⇒ toDec−warning
$a = -2.00 \wedge s = 500$
$\wedge v = 49.00 \wedge v_{safe} = 20$
$a = -2.00 \wedge s = 500$
$\wedge v = 20.02 \wedge v_{safe} = 20$

(a) Counterexample in the form of a transformation sequence $G_0 \Rightarrow_{\text{toSteady-const-warning}} G_1 \Rightarrow_{\text{toDec}} G_2$

(b) Counterexample in the form of a transformation sequence $G_0 \Rightarrow_{\text{toDec-warning}} G_1 \Rightarrow_{\text{toDec}} G_2$

Figure 9.8. – Counterexamples for $n = k = 2$

However, to improve readability, the counterexamples shown here are (minimal) transformation sequences using attributed graphs, which were manually instantiated from the s/t-pattern sequences.

Figure 9.8 shows two representative counterexamples for $n = k = 2$. In Figure 9.8(a), we can see that there is not nearly enough time to decrease the shuttle's velocity: it passes the warning with a velocity close to the target velocity of 20 m/s, but cannot decelerate because the following track t2 requires constant velocity. After passing that track, acceleration is set to −2 m/s – but only after the shuttle has already reached the signal.

Even if there is no track annotated with const, there is not enough time: Figure 9.8(b) shows that even if the shuttle sets its accelearion to −2 m/s, the one remaining track before the signal is not enough if the shuttle's velocity when reaching t1 is too high.

There is a similar transformation sequence $G_0 \Rightarrow_{\text{toSteady-const-warning}} G_1 \Rightarrow_{\text{toDec}} G_2 \Rightarrow_{\text{toDec}} G_3$ for $n = k = 3$, which is not shown here. Figure 9.9 depicts a counterexample for $n = k = 4$: here, two tracks with a const flag prevent the shuttle from decelerating enough. Increasing the minimum distance between tracks that require constant velocity (here: at least one track in between) would help to establish the forbidden pattern as a 4-inductive invariant for $n = 4$. However, we will also find that it is a 5-inductive invariant for $n = 5$. △

In contrast to the other experiments, the examples discussed above were verified on a server system running Linux with 32 GB of main memory (although Java heap space was limited to 1 GB) and an Intel E5-2640 processor with six cores at 2.5 GHz. The server was also running Java 8 and Eclipse 4.5.2 (Mars.2). The verification results are shown in Table 9.8. Leaving aside the uncertainty that comes with the lack of formal proof for the soundness of the attribute-related parts of the procedure, we can apparently verify k-inductive invariants for attributed transformation sequences using symbolic graph rules – at least for the current example. The verification time required rises more steeply in comparison to non-attributed cases, but, at least up to shuttle-attributes-5, is still acceptable.

We have left out the base case for lack of support in the implementation; for now, we can require a start configuration constraint that forbids a shuttle on a const track and requires a starting position for a shuttle of at least 6 tracks away from a signal.

Allowing attributes and attribute constraints in (symbolic) graph rules and graph con-

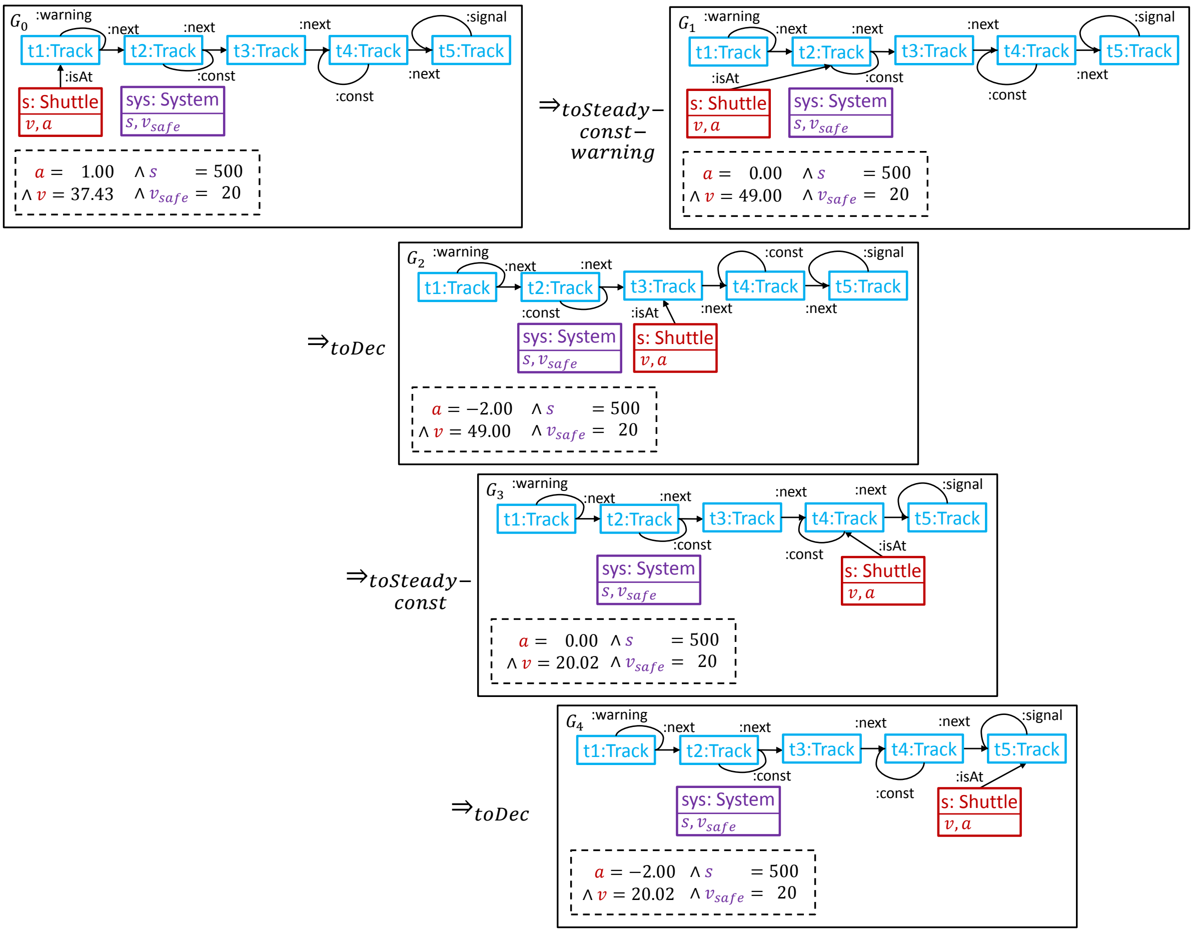

G_0
:warning
:next
:next
:next
:signal
t1:Track
t2:Track
t3:Track
t4:Track
t5:Track
:isAt
:const
:next
:const
s: Shuttle
v, a
sys: System
s, v_{safe}
$a = 1.00 \wedge s = 500$
$\wedge v = 37.43 \wedge v_{safe} = 20$
$\Rightarrow toSteady-const-warning$
G_1
$a = 0.00 \wedge s = 500$
$\wedge v = 49.00 \wedge v_{safe} = 20$
$\Rightarrow toDec$
G_2
$a = -2.00 \wedge s = 500$
$\wedge v = 49.00 \wedge v_{safe} = 20$
$\Rightarrow toSteady-const$
G_3
$a = 0.00 \wedge s = 500$
$\wedge v = 20.02 \wedge v_{safe} = 20$
$\Rightarrow toDec$
G_4
$a = -2.00 \wedge s = 500$
$\wedge v = 20.02 \wedge v_{safe} = 20$

Figure 9.9. – Counterexample for $n = k = 4$

Table 9.8. – Systems shuttle-attributes-n and k-inductive invariant checking with attributes

					k-invcheck ith attributes		
					-		
Example ($\lvert\mathcal{R}\rvert$/ $\lvert\mathcal{F}\rvert$/ $\lvert\mathcal{H}\rvert$)	n	k	cy^{+}	cy^{-}	time (s)	# ce	result
shuttle-attributes-2 (9/2/15)	2	2	10	21	< 1	6	F
shuttle-attributes-3 (9/2/15)	3	3	12	62	2	12	F
shuttle-attributes-4 (9/2/15)	4	4	14	315	12	8	F
shuttle-attributes-5 (9/2/15)	5	5	16	2433	63	0	T

straints, even if certain restrictions apply, opens up new application scenarios for the restricted approach. However, as long as soundness for the separate consideration of attribute-related elements has not been shown, we cannot be sure about verification results. While counterexamples can be validated by hand, absence of counterexamples cannot be confirmed without significant effort – effort that, if invested, might make automated execution redundant. Still, this example serves as a demonstration of the significance of attributes and attribute constraints in verification with k-inductive invariants. It also shows that, in principle, their support is feasible.

9.4. Case Study: List

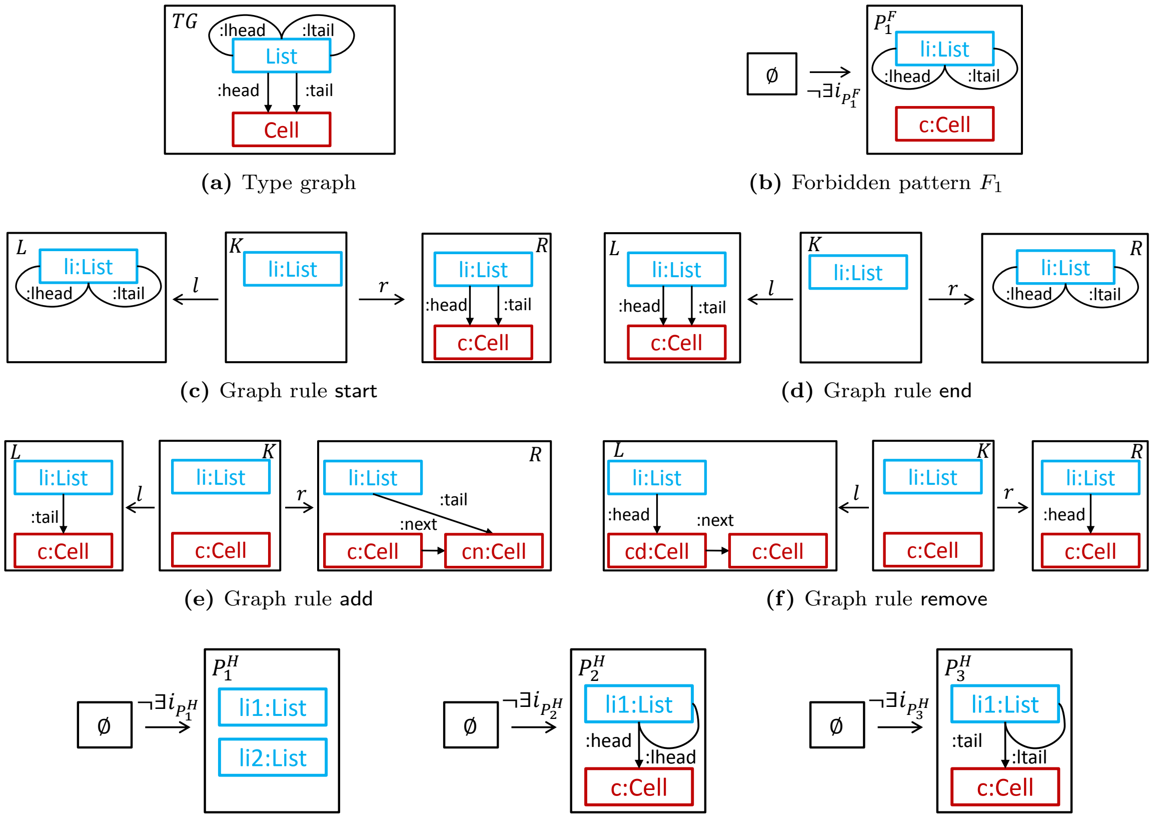
In this section, we will discuss the list case study, which was used by Steenken to demonstrate verification of infinite-state graph transformation systems by abstraction [Ste15].

Example 9.4 (system list)**.** Originally, the example was used with labeled graphs; here, it has been converted to graphs typed over a type graph TG (Figure 9.10(a)). The composed forbidden pattern $\mathcal{F} = \neg F_1$ consists of one forbidden pattern $F_1 = \exists i_{P_1^F}$ (Figure 9.10(b)). Graph rules in $\mathcal{R}$ can start (Figure 9.10(c)) an empty list by adding a cell and end (Figure 9.10(d)) a list by removing its last cell. Cells can also be added (Figure 9.10(e)) to the end of the list and removed (Figure 9.10(f)) from the top of the list. The list points to its top and end by edges head and tail. In the original example, both head and tail could point from an empty list to itself or from a non-empty list to cells; here, because we use typed graphs, we have to separate both as different edge types. Since we only have a single list and need certain other cardinality constraints for the type graph, we also have a composed guaranteed pattern $\mathcal{H} = \neg H_1 \wedge \neg H_2 \wedge \neg H_3$ (Figures 9.10(g)-9.10(i)). While there are other valid type graph constraints, they will not be required here.

In the original example [Ste15], there is also a start graph G_0 that contains the left side of start: an empty list with both lhead and ltail pointing to itself. Here, we do not usually specify start graphs. We could use start configuration patterns to describe this start graph by requiring the absence of a cell and requiring every list to have either a cell or ltail and lhead edges pointing to itself. However, this will not be the focus here.

Using abstraction and the approach by Steenken, Wehrheim, and Wonisch [Ste15, SWW11, SW11], it can be established [Ste15] that the state space of $GG = (GTS, G_0)$ does not contain a violation of $\mathcal{F}$. Here, we attempt to show that $\mathcal{F}$ is a 1-inductive invariant of $\mathcal{R}$ under $\mathcal{H}$. Verification with 1-invcheck and rule applicability (appl.) takes less than a second. Unfortunately, we get a negative result. The only counterexample is shown in Figure 9.11; for the sake of brevity, negative application conditions requiring the absence of edges adjacent to cell c (so that it can be deleted) are not depicted.

This counterexample is not a false negative: $\neg F_1$ is simply not a 1-inductive invariant for GTS under $\mathcal{H}$. It is an operational invariant of $GG = (GTS, G_0)$ – we will never encounter the situation represented in S_1 because a cell can never be isolated: it is created only in conjunction with a tail edge and, if it is not the first cell, a next edge. A cell will only lose all of its connecting edges if it is deleted, including next edges. Using shape abstraction, the knowledge about the creation of cells in the state space can be taken into account – with 1-inductive invariant checking, we would have to somehow encode it into additional constraints to be verified. Unfortunately, this would include following the (transitive) next edges to the head or tail of the list; since the list may have an arbitrary length, we cannot do that (by inductive invariant checking). Here, lack of information about the state space and the reachability of states become apparent as one of the drawbacks of verification with inductive invariants in certain situations. Applying k-induction for a value larger than 1 will not help either – a



(a) Type graph

(b) Forbidden pattern F_1

(c) Graph rule start

(d) Graph rule end

(e) Graph rule add

(f) Graph rule remove

(g) Guaranteed pattern $\neg H_1$

(h) Guaranteed pattern $\neg H_2$

(i) Guaranteed pattern $\neg H_3$

Figure 9.10. – Type graph TG, forbidden pattern F_1, rules $\mathcal{R}$, and composed guaranteed pattern $\mathcal{H} = \neg H_1 \wedge \neg H_2 \wedge \neg H_3$ for list

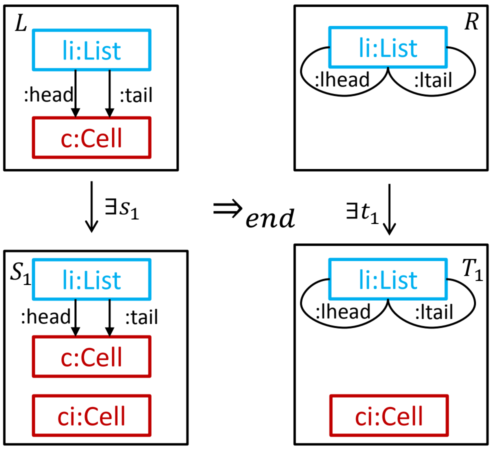
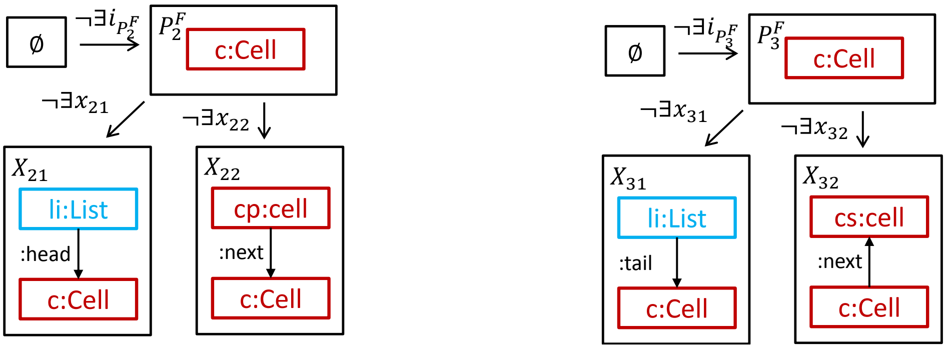


Figure 9.11. – Counterexample $src_1 \Rightarrow_{\mathsf{end}} tar_1$



(a) Pattern $F_2 = \exists(i_{P_2^F}, \neg\exists x_{21} \wedge \neg\exists x_{22})$

(b) Pattern $F_3 = \exists(i_{P_3^F}, \neg\exists x_{31} \wedge \neg\exists x_{32})$

Figure 9.12. – Forbidden patterns F_2 and F_3

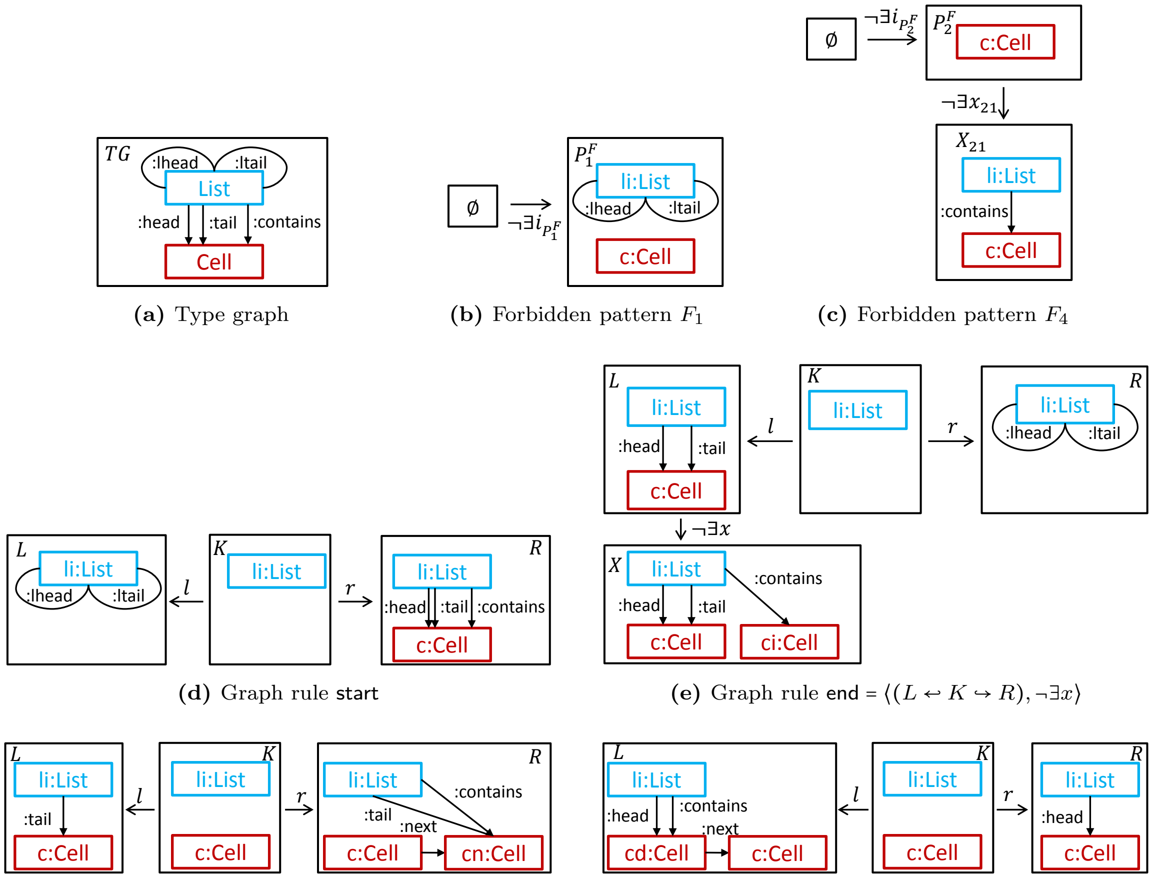
counterexample might simply be the one shown before, but with an additional application of start before it.

That said, we can verify other interesting properties with inductive invariant checking (and the restricted approach). Two examples are shown in Figure 9.12: every cell is either a list's head or has a predecessor (Figure 9.12(a)) and every cell is either a list's tail or has a successor (Figure 9.12(b)). These properties are 1-inductive invariants, they are obviously valid in the start graph and we can even show that an equivalent formulation of the start graph as a composed sart configuration pattern implies both properties.

We may be tempted to add the original forbidden pattern (conjunctively) to those two forbidden patterns as part of a new composed forbidden pattern $\mathcal{F}' = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$ and apply 1-induction again. We will still get $src_1 \Rightarrow_{\mathsf{end}} tar_1$ as a counterexample. However, we can see that certain satisfying transformation sequences $G_0 \Rightarrow_{\mathsf{end}} G_1$ will have G_0 violate $\neg F_2$ and $\neg F_3$. Take, for instance, the graph S_1 directly from the source pattern: since it contains L and its embedding in S_1, it satisfies $src_{1|\varnothing}$; however, it has a cell without a successor, a predecessor, and head and tail edges. In general, this looks like a typical case where implication with composed graph patterns can be applied: we can use, for instance, F_2 to conclude the existence of a preceding cell to ci (a second head edge should be prohibited by a type graph constraint in the composed guaranteed pattern). However, this will only transfer the problem to the new cell, which does not have a predecessor or a head edge either. Repetition of this procedure will not lead to termination: the counterexample is indeed a false negative (given $\mathcal{F}' = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$, not the original $\mathcal{F} = \neg F_1$) because we cannot find a (finite) graph satisfying $src_{1|\varnothing}$, and $\mathcal{F}'$ at the same time. However, we cannot prove it by using the restricted (or general) approach to 1-inductive invariant checking; again, this would require symbolically following the next edges to the list's head or tail.

There is a way out of this: we can require cells to be connected to their containing list and create and delete such contains edges along with the creation and deletion of cells. If we change the end rule such that it is applicable only if the list does not contain more than one cell, we can verify two (conjunctively joined) forbidden patterns $\mathcal{F} = \neg F_1 \wedge \neg F_4$ as a 1-inductive invariant via 1-invcheck with appl. and impl. (in less than a second). We need both forbidden patterns: every cell is contained in the list (Figure 9.13(c)) and the original forbidden pattern $\neg F_1$ (Figure 9.13(b)). (We can also verify $\mathcal{F}'$ from before.) The adapted type graph, rules, and forbidden patterns of the new system list-containment are depicted in Figure 9.13. The composed guaranteed pattern remains the same; while we could add cardinality constraints for contains, those are not necessary for verification. △

Verifying this example required significantly more specification and verification effort than



(a) Type graph

(b) Forbidden pattern F_1

(c) Forbidden pattern F_4

(d) Graph rule start

(e) Graph rule end = $\langle(L \hookleftarrow K \hookrightarrow R), \neg\exists x\rangle$

(f) Graph rule add

(g) Graph rule remove

Figure 9.13. – Type graph TG, forbidden pattern F_1, and rules $\mathcal{R}$ for list-containment

expected: we have slightly changed one rule and added an edge type and a new forbidden pattern. While the behavior of the list arguably remains the same and we get the same result, this approach may not always be applicable or feasible for more complex systems. In summary, the benefit of inductive invariant checking of disregarding state spaces and reachability (beyond the bounds of k-induction) comes with disadvantages: sometimes, we cannot show properties that rely on the state space or its larger history (here: the transitive closure of next) – or we can only show them by experimenting and encoding additional information into the system and the composed patterns relevant for verification.

9.5. Concurrent Behavior

In the shuttle case study, we have used a guaranteed pattern that forbids the existence of two shuttles. If we drop this restriction, applying 2-inductive invariant checking becomes challenging. Here, we will get valid counterexamples with two shuttles in the following sequence: first, one shuttle (legally) moves somewhere else, second, another shuttles moves in speed mode fast to a switch. Of course, that second shuttle would not have been able to arrive in speed mode fast on its current track (with a switch directly ahead). Usually, we can detect this by extending k – however, given concurrent behavior happending somewhere else, we do not learn about the violating shuttle's actions leading to the violation. In essence, although movement of multiple shuttles is meant to happen concurrently, this is not reflected in the counterexamples.

This problem is partly inherent in the nature of graph transformation systems and transformation sequences: without further considerations, they describe sequential application of graph rules, not concurrent behavior. This aspect is not relevant for k-inductive invariant checking with $k = 1$ because it only considers one rule application as opposed to a longer sequence.

When it occurs, there are several possible solutions to the problem:

One behavioral entity. We can restrict the number of behavioral entities to exactly one – as we have done for the shuttle case study before. Then, the execution of rules in s/t-pattern sequences will indeed happen in a sequential fashion. We can still argue that the system is safe even for mutliple behavioral entities (shuttles) if they are independent of each other. Here, this is the case because the application of behavioral rules and the compoesd patterns do not depend on connections or relations between shuttles. In general, this means arguing that a behavioral entity's individual behavior is safe. In other words, by restricting the number of behavioral entitites to one, we anaylze an individual state space (or, given an infinite number of start graphs, individual state spaces) for each behavioral entity. Non-behavioral entities (e.g. the track topology) remain stable and isomorphic in all states of all state spaces. If the composed forbidden pattern was established as an operational invariant for the system with only one behavioral entity, this holds for all such state spaces. Then, given independence of behavioral entities, operational invariance still holds for the cross product of all state spaces. This, then, is equivalent to the state space without restrictions on behavioral entity. We can apply this argument to our shuttle protocol.

Upper bound on behavioral entities. This idea is based on the first; there may be more than one, but a finite limited number of behavioral entities. This will still lead to the problem sketched above; however, after reaching the limit of behavioral entities, increasing the value of k in k-indudctive invariant checking will only consider rule applications for these entities, not create new counterexamples with a rising number of entities. However, this approach usually needs to be combined with one of the ideas following below.

Locality of behavioral entities. This is somehow similar to the argument of independence between behavioral entities. If k-indcutive invariant checking produces counterexamples where each behavioral entity operates on a graph component that is (within the local context of the counterexample) not connected to the other entities' components, we may not need to consider them. If we are also able to exclude behavioral entities in close proximity, we can again rely on the individual analysis of entities. For example, we can argue that shuttles in different and (in a local context) unconnected parts of the topology should not influence each other's behavior. Furthermore, we may be able to separately verify that shuttles are never within a certain distance of each other, given appropriate rules. For some systems, this minimum distance may also be an external assumption of the specification. Then, counterexamples with entitites on locally unconnected components can be discarded; also, entities on connected parts within the minimum distance do not need to be considered.

Enforcing round-robin execution in sequences. If behavior of all behavioral entities is supposed to happen concurrently, we can discard all counterexamples where the same entity performs two actions without all other entities (that appear in the counterexample) having acted in between. Of course, this only works if entities are forced to take an action. For example, if shuttles may simply stand still indefinitely, this idea is not applicable – unless we argue that inactive entities should not be considered in counterexamples (because they do not influence other entities' behavior, see above).

Combining rules and rule applications. We can combine the application of two behavioral rules (to the same or different entities) by using *E-concurrent rules* [EGH+14]. An E-concurrent depends on how the rules overlap (i.e. influence and depend on) each other (in a graph E); examining all possibilities requires investigating all possible E-concurrent rules. Then, we can apply k-inductive invariant checking with this newly created set of rules. Thus, execution of behavior within an E-concurrent rule is truly concurrent. If there are more than two behavioral entities, further combinations of rules may be required, possibly up to the upper bound on the number of entities.

E-concurrent rules were used for the verification of behavior preservation (cf. Section 9.2) for the phase of model semantics. (However, the goal was to express the alternating execution of behavior in source and target models to verify bisimilarity, not to apply k-inductive invariant checking for $k \geq 2$.)

A more detailed discussion of the impact of concurrent behavior and an automated implementation of possible solutions is beyond the scope of this thesis. Implementation of the solutions as part of the invariant checker would also require their formalization, formal proof, and evaluation. Note that the possible existence of multiple behavioral entities rarely causes problems for $k = 1$ and does not always lead to the problem sketched here even for $k \geq 2$. For now, where it does, we need to handle it on a case-by-case basis.

9.6. Discussion and Conclusion

In summary, the approach can be successfully and meaningfully applied for the behavior preservation and shuttle protocol case studies. With some restrictions and interaction, we can also apply it for the list case study. We have seen in Chapters 6 and 7 that the approach and its extensions are sound (**Appl.-soundness**) and terminate (**Appl.-termination**) by design; both has been confirmed for the case studies. Furthermore, we have the following observations with respect to the initial verification problem:

State space restrictions. Composed guaranteed patterns have turned out to be useful to formulate type graph constraints not otherwise accessible in the approach and implementation. In particular, they were required for the behavior preservation case study to exploit the results of verification of the model transformation phase (e.g. equiv-mt) in the phase concerned with model semantics (equiv-sem). Without this structure, verification would not have succeeded; or would have required iteration over the composed forbidden pattern and user interaction.

Furthermore, they were used to model a single fault assumption in shuttle-single-fault-unsafe and shuttle-single-fault-safe.

Infinite number of start graphs. We have shown that, for the shuttle protocol, $k-1$-bounded backward model checking is applicable to show validity of the composed forbidden pattern (under the composed guaranteed pattern) in the (here) 1-bounded state space of all graph grammars induced by the graph transformation system and a composed start configuration pattern. In combination with the 2-inductive invariant established for shuttle-safe and shuttle-single-fault-safe, our inductive argument lets us conclude operational invariance of the safety property for all (infinite) induced graph grammars. For the systems shuttle-unsafe and shuttle-single-fault-unsafe, we can use $k-1$-bounded backward model checking to find symbolic counterexamples for error traces from possible start graphs to a violation.

Reduction of false negatives. For shuttle-single-fault-safe, we needed a 2-inductive invariant in the inductive step of our inductive argument; the required property was not a 1-inductive invariant. We have also seen that simulation of k-induction with the restricted approach to 1-inductive invariant checking is not feasible (Section 9.3.2). Furthermore, performing k-induction with tools like Enforce requires changing the program specification and appears impractical for performance reasons. Finally, we have seen the relevance of forward propagation and implication with composed patterns with respect to the reduction of false negatives (Sections 9.2 and 9.3.1).

In all cases, we could eliminate potential false negatives if forward propagation and implication with composed pattern were used. Thus, we have a reasonable degree of completeness (**Appl.-deg.completeness**) for the given examples. Likewise, performance is not an issue for the shuttle protocol and the list case study. For refine-trans it is still within reasonable bounds. Thus, we can argue for satisfaction of **Appl.-performance**.

Of course, it is difficult to generalize from these case studies. Given the structure of the approach and its implementation, it can be expected to be applicable to systems with similarly-sized rules and patterns, similar types of negative application conditions, and similar values of k. Issues with performance may possibly be addressed by using appropriate configurations and extensions, most importantly preprocessing, partial negative application conditions, skipping file output of counterexamples, and terminating after returning the first counterexample. Possible issues with false negatives can be solved with forward propagation or implication with composed patterns.

While the 17 minutes required for verification of refine-trans are still acceptable, addition of any further nodes to rules and the negative application condition in this example's patterns may increase verification time beyond reasonable bounds. In Section 9.3.4, we have seen that just one or two nodes can make a drastic difference in complexity – and in cases like refine-trans, partial negative application conditions will not offset this increase. This is a risk no matter the approach used; invariant checking is particularly suspectible for large negative application conditions. If all else fails, we may attempt to (equivalently) modify the system specification:

connecting nodes may be represented by edges; multiple nodes could be summed up in a single node. However, we would always need to argue that this modifications indeed preserve the original system's behavior, which is often non-trivial.

An important question remains the choice of k when we want to verify a system. For the shuttle case study, we can guess the expected value of $k = 2$ for shuttle-safe and shuttle-single-fault-safe because negative application conditions in rules check for switches two tracks ahead. In other cases, this may not be so obvious; also, if a k-inductive invariant cannot be established, we would need to decide when to terminate. If we are not sure about a certain value for k, we can always start with $k = 1$ and iteratively increase its value, possibly examining counterexamples along the way. (We have seen in Section 9.3.2 that this is different from using 1-induction to simulate k-inductive invariang checking.)

For $k-1$-bounded backward model checking, we have seen a steeper increase in required runtime than for k-induction – at least for the examples used here (Section 9.3.1). Bounded backward model checking as used here is primarily meant to verify validity of a composed forbidden pattern in a bound state spaces for lower values of k. While we can use it to create symbolic error traces from possible start graphs to violations (e.g. to gain insight into a system), its application is impractical once the paths reach a certain length (such as $k = 6$ for shuttle-unsafe). Hence, the current implementation of $k-1$-bounded backward model checking is not meant to compete with state-of-the-art model checkers (such as GROOVE [GdMR+12], Henshin [ABJ+10]) capable of exploring finite state spaces for a given start graph or a low number of start graphs. Likewise, $k-1$-bounded model checking cannot, by itself, explore infinite state spaces (such as by abstraction [Ste15]) – our capability of arguing for systems with infinite state spaces comes from the combined indcutive argument of $k-1$-bounded backward model checking (base case) and k-inductive invariant checking (inductive step).

We have also seen limitations of the approach's applicability: sometimes, desired safety properties are – by themselves – not k-inductive invariants although they are operational invariants. This may happen, for instance, if properties of nodes and edges stay relevant beyond a local context and throughout the whole state space. For example, in the list case study, cells are always created with connecting edges – if we encounter a cell in a counterexample, this information about its original creation is not easily available. Sometimes, this knowledge may be available because of type graph constraints (i.e. as guaranteed patterns); in other cases, we might be able to guess such properties, explicitly verify them, and then succeed to verify the original composed forbidden pattern. If not or if this is too much effort, we can try verification approaches and tools focusing on state spaces and reachability – GROOVE [GdMR+12], Henshin [ABJ+10], abstraction and shape analysis [Ste15].

As mentioned before (and reiterating results from Chapter 8), the restricted formal model may also require changing systems to establish conformity; other systems may be out of bounds because of the limitations on, e.g., nested graph constraints. The same applies to graph programs with a control program beyond sets of graph rules with composed negative application conditions and priorities; here, we might need Enforce. Sometimes, we may wish to verify temporal properties or formulas in monadic second-order logic; both are beyond the capabilities of even the general approach (and Enforce). Finally, systems with concurrent behavior and multiple units with behavioral capabilites (such as shuttles) may lead to problems for the tool's application (cf. Section 9.5).

That said and in conclusion, verification with k-inductive invariant checking and $k-1$-bounded backward model checking has successfully been applied to relevant examples and will likely be applicable to systems of similar characteristics and expected values of k.

10. Conclusion and Outlook

This chapter will first summarize the results of this thesis, particularly with respect to its contribution. Second, we will discuss possibilities for further extension of the approach and tool and outline future work.

10.1. Conclusion and Contribution

In order to summarize this thesis's contribution, we recall the generic verification problem brought up in Chapter 1:

Verification Problem 1.3. *Given a set of systems defined by a system metamodel, a set of initial states, specification of system behavior, and restrictions on the state space and given a set of safety properties, does every state in the restricted state spaces of all systems satisfy the safety properties?*

With respect to this verification problem, this thesis pursued the following contribution:

Formal-general – the description and justification of a formal and symbolic approach solving Verification Problem 1.3 by verification of graph transformation systems with k-induction, called the *general approach*,
Formal-restricted – the description and justification of a formal and symbolic approach solving Verification Problem 1.3 by verification of graph transformation systems with k-induction for the restricted formal model, called the *restricted approach*,
Impl.-restricted – the implementation of the approach described by **Formal-restricted** as an automated procedure,

where the formal approach (**Formal-restricted**) and implementation (**Impl.-restricted**) should be applicable to meaningful scenarios, and, in their application, provide a positive result or meaningful symbolic counterexamples and have the following properties:

Appl.-soundness – soundness,
Appl.-termination – termination,
Appl.-deg.completeness – a reasonable degree of completeness,
Appl.-performance – reasonable performance.

Based on the foundations in Chapters 2 and 3 and the extensions of Chapter 4, we were able to map Verification Problem 1.3 to the world of graph transformation systems:

Verification Problem VP.1g. *Given a graph transformation system $GTS = (TG, \mathcal{R})$ and graph constraints $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{H}$ with $\mathcal{S} \models \mathcal{F}$, does every graph grammar $GG \in \text{IND}(GTS, \mathcal{S})$ have $\mathcal{F}$ as an operational invariant under $\mathcal{H}$?*

This, then, was the problem solved in Chapter 5. Theorems T.1g (p. 85), T.2g (p. 96), T.3g (p. 101), T.4g (p. 104), and related constructions, theorems, and lemmas made up the general approach – or **Formal-general**.

Given the challenges coming with the general approach with respect to computational effort and complexity, we refined the general approach, starting with the verification problem:

Verification Problem VP.1r. *Given a graph transformation system $GTS = (\mathcal{R}, TG)$ with rules of the form $b_i = \langle (L_i \hookleftarrow K_i \hookrightarrow R_i), ac_{L_i}, \text{true}\rangle$ for composed negative application conditions ac_{L_i} and composed graph patterns $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{H}$ with $\mathcal{S} \models \mathcal{F}$, does every graph grammar $GG \in \text{IND}(GTS, \mathcal{S})$ have $\mathcal{F}$ as an operational invariant under $\mathcal{H}$?*

Based on that, we refined the formal model, symbolic encoding, and constructions of the general approach in Chapter 6. The result, mainly in the form of Theorems T.1r (p. 130), T.2r (p. 143), T.3r (p. 149), and T.4r (p. 153), is the restricted approach to k-inductive invariant checking and $k-1$-bounded backward model checking: **Formal-restricted**. By inductive argument (Lemma L.1 (p. 65)), it solves Verification Problem VP.1r. The approach satisfies **Appl.-soundness** and **Appl.-termination** by construction and design.

While the general approach considered Verification Problem VP.1g on a formulaic level only, the restricted approach also comes with an implementation (**Impl.-restricted**) of automated k-inductive invariant checking and $k-1$-bounded backward model checking. Its drawbacks, most importantly incompleteness, have been addressed in a number of extensions (Chapter 7): forward propagation and implication with composed patterns aims to address incompleteness (and partly, performance), partial negative application conditions target computational effort.

The restricted approach was shown to be applicable to a number of examples and case studies in Chapter 9 – including cases where application of other approaches was problematic. We could successfully verify the invariance of safety properties as state properties in states of state spaces of induced graph grammars. For systems with violations of these safety properties, meaningful counterexamples could be used to attempt to repair the system. The evaluation has shown a sufficient degree of completeness (**Appl.-deg.completeness**) of the approach for the examples used – and has demonstrated the effectiveness and impact of the extensions, particularly forward propagation and implication with composed graph patterns. We can also suspect a similarly reasonable degree of completeness for similar systems. We have also seen that we can work around limitations of the approach (cf. Section 9.4) in some cases. As a static and symbolic verification approach that can yield results for infinite sets of graph grammars with infinite state spaces given a fixed graph transformation system, performance (**Appl.-performance**) was also reasonable for the examples used.

The existence of numerous complementary (and somtimes competing) approaches to verification of graph transformation systems leaves little doubt about the complex and challenging nature of the problem. Approaches based on explicit state space construction often exhibit impressive results even for large state spaces and offer verification of temporal properties in addition to state properties. However, they are limited with respect to the size of state spaces and are not applicable for infinite state spaces. Using abstraction to represent infinite state spaces requires striking a balance between a meaningful concept of abstraction that preserves important information and a sufficient reduction of said infinite or large state spaces. Finally, symbolic approaches (such as k-inductive invariant checking) that abstain from exploring state spaces explicitly or by abstraction maneuver between efficient verification of relevant systems with infinite state spaces and the risk of high computational effort, false negatives, undecidable problems, and limitations on what can and cannot be specified.

All approaches in these classes may or may not address a second degree of complexity or infinity regarding the number of initial states and graph grammars analyzed in one application. They also need to decide on the level of expressive power allowed in specifications (and hence, verification). Finally, each class comes with its own challenges with respect to performance and efficiency: exploration of explicit state spaces may lead to exponential growth with the number of rules; algorithms that reason with and about graph constraints may have exponential complexity with the number of nodes and edges, i.e. the size of graphs.

Given the different characeteristics of graph transformation systems and graph grammars with respect to start configurations, behavior, and state spaces, there is, as of yet, no single approach or tool that has been established as a default choice – which is probably fortunate for the variety and diversity of research on graph transformation systems and their verification. Thus, the approach and implementation discussed in this thesis is not meant to replace any existing tool (except for its previous incarnations), but to fit in the landscape of verification approaches for graph transformation systems. In summary, the approach and tool is, in theory, applicable for systems (graph grammars) where

- start graphs can be characterized by a limited form (composed pattern) of graph constraint (or there is a finite amount of start graphs),
- safety properties to be verified can be specified by a composed pattern,
- we may or may not require additional restrictions of the state space or external assumptions or information encoded by another composed pattern,
- graph rules may or not have (left) composed negative application conditions,
- where graphs in state spaces may or may not be large,
- and where the resulting state spaces may be infinite.

Application is particularly promising with respect to feasibility and completeness if

- rules are small (such as less than seven nodes per side),
- forbidden patterns – and, more importantly, their negative application conditions – are small (such as less than seven nodes per condition and pattern),
- there are few common elements between rules and forbidden patterns,
- there are established system properties with a global character (such as a limit of the numner of nodes or edges of certain types),
- there are few non-trivial negative application conditions (although their effects may vary widely),
- and if system behavior has a sequentially dependent character in the sense that rule applications affect each other (such as a shuttle changing speed modes).

Leaving aside the reversal of the points above, the approach may encounter problems if the specification does not conform and cannot be adjusted to conform to the restricted formal model and if

- patterns describe transitive properties (such as requiring a node to always have a successor),
- properties of the state space are relevant beyond their local context (when, for example, characteristics of nodes depend on how they were created);
- similarly, if behavior or forbidden patterns are detached from local context (such as isolated nodes or separate graph components),
- and if there is concurrent behavior, especially if there is an arbitrary number of elements actively exhibiting behavior (such as shuttles).

Some of this problems prevent successful application of the approach; others can be addressed if they occur. For example, earlier work [BLD+11] described verification of consistency-preserving Java refactorings with 1-inductive invariant checking. Some refactorings require comparing signatures of Java methods, which may include lists of parameters of arbitrary length. Equality of such lists, however, cannot be expressed by nested graph constraints. The solution was to introduce a number of helper rules that computed these and similar properties and rely on their information during verification of the refactoring rules. Then, if the execution of such refactorings has a similar preprocessing phase, the verification result also applies to the actual execution.

The availability of a number of extensions and configuration options for the approach's implementation enables an experienced user to cope with certain problems and avoid others. For example, preprocessing with implication for composed graph patterns (Theorem 7.40 (p. 225)) has been shown to greatly improve performance on occasion. Partial negative application conditions may have a similar effect. If verification of a system takes too long, the stop option can be used to inspect the first counterexample (if any) in order to gain some insight into the system. Modifying a system specification in meaningful ways and extending composed forbidden patterns may help k-inductive invariant checking to succeed; this was shortly sketched in Section 9.4. Since software and system development usually happens in an iterative fashion, it is no surprise that verification may sometimes benefit from following a similar path.

Often, the restricted approach and its implementation can be combined with other tools. Subproblems that do not conform to the restricted formal model might be outsourced to tools like Autograph [SLO17, SLO18] or Enforce [Pen09]. This also applies to the analysis of counterexamples with respect to whether or not they may be false negatives. Given s/t-pattern sequences, we might apply model checking to create specific transformation sequences represented by the counterexamples. Similarly, if verification is expected to take a long time, we can use explicit-state model checking to explore state spaces and eliminate counterexamples by fixing specification errors before applying k-inductive invariant checking.

10.2. Open Issues and Future Work

As is common for finding solutions for research questions – and here, verification problems – this thesis has also opened up new questions, issues, and challenges. Here, we will briefly discuss some aspects that are worthy of further discussion in future work.

Heuristics for verification problems. In the restricted approach and its implementation, there are a number of configuration options users can choose from, such as the value of k and which extensions to use. While it is possible to experiment with configurations, it would be nice to determine possibly fitting configurations by analyzing the system in question and comparing it to known heuristics. Future work might research useful metrics and their effect on configurations. Other metrics could be concerned with determining the likelihood of encountering false negatives depending on the class or nature of a system.

General approach and expressive power. With appropriate optimizations, it seems possible that an implementation of the general approach and its application are feasible for some systems. A combination of the restricted and general approaches is also an idea: if a system's specification conforms to the restricted approach, we can use that; otherwise, the system is verified with the general approach. Extensions of the restricted approach with respect to what we allow in specifications are also among obvious ideas for future work. For example, we can think of limited support for graph programs realized by analyzing s/t-pattern sequences and discarding those not conforming to the graph program. Other extensions might focus on the general approach and move from nested graph conditions to more expressive concepts.

Attributes and symbolic attributed graph transformation systems. While there is an implementation available that supports attribute constraints in graph rules and graph constraints (with respect to k-inductive invariant checking), it has not been established with the formal rigorousness employed for the general and restricted approach. Formally describing k-inductive invariant checking for restricted forms of symbolic attributed graph constraints and symbolic attributed graph transformation systems and proving its soundness is an open point.

Concurrent behavior. Finally, Section 9.5 has brought up a number of questions and ideas to solve the challenge of applying k-induction for systems with multiple behavioral entities. This, too, is an open issue that deserves to be investigated further.

This overview of open issues and future work concludes this thesis. It has addressed aspects of the greater underlying problem of formal verification for complex systems. It has provided a formal description, implementation, and evaluation of verification of graph transformation systems in a symbolic fashion – namely with induction and k-inductive invariants. Given the history and current state of software and hardware systems and their development, formal verification will remain relevant. And, given the history and current state of research on and application of graphs and graph transformation systems, formal verification of the latter will presumably remain relevant as well.

Publications of the author in the context of this thesis

[1] Johannes Dyck and Holger Giese. Inductive Invariant Checking with Partial Negative Application Conditions. In Francesco Parisi-Presicce and Bernhard Westfechtel, editors, *Graph Transformation*, volume 9151 of *LNCS*, pages 237–253, Cham, 2015. Springer.

[2] Johannes Dyck and Holger Giese. Inductive Invariant Checking with Partial Negative Application Conditions. Technical Report 98, Hasso Plattner Institute, University of Potsdam, 2015.

[3] Johannes Dyck and Holger Giese. k-Inductive Invariant Checking for Graph Transformation Systems. In Juan de Lara and Detlef Plump, editors, *Graph Transformation*, volume 10373 of *LNCS*, pages 142–158, Cham, 2017. Springer.

[4] Johannes Dyck and Holger Giese. k-Inductive Invariant Checking for Graph Transformation Systems. Technical Report 119, Hasso Plattner Institute, University of Potsdam, 2017.

[5] Johannes Dyck, Holger Giese, and Leen Lambers. Automatic Verification of Behavior Preservation at the Transformation Level for Relational Model Transformation. Technical Report 112, Hasso Plattner Institute, University of Potsdam, 2017.

[6] Johannes Dyck, Holger Giese, and Leen Lambers. Automatic verification of behavior preservation at the transformation level for relational model transformation. *Software & Systems Modeling*, 18(5):2937–2972, 2019.

[7] Johannes Dyck, Holger Giese, Leen Lambers, Sebastian Schlesinger, and Sabine Glesner. Towards the Automatic Verification of Behavior Preservation at the Transformation Level for Operational Model Transformations. In Jürgen Dingel, Sahar Kokaly, Levi Lúcio, Rick Salay, and Hans Vangheluwe, editors, *Analysis of Model Transformations*, volume 1500 of *CEUR Workshop Proceedings*, 2015.

Bibliography

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 121–135, Berlin/Heidelberg, 2010. Springer.

[BBG+06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proceedings of the 28th International Conference on Software Engineering*, pages 72–81, New York, 2006. ACM.

[BBKR08] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink. A Modal-Logic Based Graph Abstraction. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, pages 321–335, Berlin/Heidelberg, 2008. Springer.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[BCK08] Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206(7):869–907, 2008.

[BG08a] Basil Becker and Holger Giese. Incremental Verification of Inductive Invariants for the Run-Time Evolution of Self-Adaptive Software-Intensive Systems. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 33–40, Piscataway, 2008. IEEE Computer Society.

[BG08b] Basil Becker and Holger Giese. On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In *11th IEEE Sympoisum on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 203–210. IEEE Computer Society, 2008.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, 2008.

[BKK+12] Iovka Boneva, Jörg Kreiker, Marcos Kurbán, Arend Rensink, and Eduardo Zambon. Graph Abstraction and Abstract Graph Transformations (Amended Version). Technical report, University of Twente, 2012.

[BLD+11] Basil Becker, Leen Lambers, Johannes Dyck, Stefanie Birth, and Holger Giese. Iterative Development of Consistency-Preserving Rule-Based Refactorings. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 123–137, Berlin/Heidelberg, 2011. Springer.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking.* Springer, 2018.

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software Verification Using k-Induction. In Eran Yahav, editor, *Static Analysis*, volume 6887 of *LNCS*, pages 351–368, Berlin/Heidelberg, 2011. Springer.

[Dyc12] Johannes Dyck. Increasing expressive power of graph rules and conditions and automatic verification with inductive invariants. Master's thesis, Hasso Plattner Institute, University of Potsdam, 2012.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages, and Tools*. World Scientific, River Edge, 1999.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Secaucus, 2006.

[EGH$^+$14] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. part 1: Parallelism, concurrency and amalgamation. *Mathematical Structures in Computer Science*, 24(4), 2014.

[EH86] Hartmut Ehrig and Annegret Habel. Graph Grammars with Application Conditions. In Grzegorz Rozenberg and Arto Salomaa, editors, *The Book of L*, pages 87–100. Springer, Berlin/Heidelberg, 1986.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *LNCS*, pages 161–177, Berlin/Heidelberg, 2004. Springer.

[GdMR$^+$12] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using groove. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.

[GHE14] Ulrike Golas, Annegret Habel, and Hartmut Ehrig. Multi-amalgamation of rules with application conditions in $\mathcal{M}$-adhesive categories. *Mathematical Structures in Computer Science*, 24(4), 2014.

[GL12] Holger Giese and Leen Lambers. Towards Automatic Verification of Behavior Preservation for Model Transformation via Invariant Checking. In Hartmut Ehrig, Gregor Engels, Hans Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 7562 of *LNCS*, pages 249–263, Berlin/Heidelberg, 2012. Springer.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, volume 2030 of *LNCS*, pages 230–245, Berlin/Heidelberg, 2001. Springer.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest Preconditions for High-Level Programs. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 4178 of *LNCS*, pages 445–460, Berlin/Heidelberg, 2006. Springer.

[HSE11] Christian Heinzemann, Julian Suck, and Tobias Eckardt. Reachability analysis on timed graph transformation systems. *Electronic Communications of the EASST*, 32, 2011.

[Ken02] Stuart Kent. Model Driven Engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *LNCS*, pages 286–298, Berlin/Heidelberg, 2002. Springer.

[KG12] Christian Krause and Holger Giese. Probabilistic Graph Transformation Systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 7562 of *LNCS*, pages 311–325, Berlin/Heidelberg, 2012. Springer.

[KK06] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 197–211, Berlin/Heidelberg, 2006. Springer.

[KK08] Barbara König and Vitali Kozioura. Augur 2 – A New Version of a Tool for the Analysis of Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 211:201–210, 2008.

[KS12] Barbara König and Jan Stückrath. Well-structured graph transformation systems with negative application conditions. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 7562 of *LNCS*, pages 81–95, Berlin/Heidelberg, 2012. Springer.

[KS14] Barbara König and Jan Stückrath. A General Framework for Well-Structured Graph Transformation Systems. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, volume 8704 of *LNCS*, pages 467–481, Berlin/Heidelberg, 2014. Springer.

[KS17] Barbara König and Jan Stückrath. Well-structured graph transformation systems. *Information and Computation*, 252:71–94, 2017.

[MGK17] Maria Maximova, Holger Giese, and Christian Krause. Probabilistic Timed Graph Transformation Systems. In Juan de Lara and Detlef Plump, editors, *Graph Transformation*, volume 10373 of *LNCS*, pages 159–175, Cham, 2017. Springer.

[Mil89] Robin Milner. *Communication and concurrency.* Prentice Hall, New York, 1989.

[Nic16] Christian Nicolai. Using exchangeable constraint solvers for invariant checking on attributed graph transformation systems. Master's thesis, Hasso Plattner Institute, University of Potsdam, 2016.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA Environment. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 742–745, New York, 2000. ACM.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, Berlin/Heidelberg, 2002.

[NSM03] Manfred Nagl, Andreas Schürr, and Manfred Münch, editors. *Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *LNCS*. Springer, Berlin/Heidelberg, 2003.

[OL10a] Fernando Orejas and Leen Lambers. Delaying Constraint Solving in Symbolic Graph Transformation. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Graph Transformations*, volume 6372 of *LNCS*, pages 43–58, Berlin/Heidelberg, 2010. Springer.

[OL10b] Fernando Orejas and Leen Lambers. Symbolic attributed graphs for attributed graph transformation. *Electronic Communications of the EASST*, 30, 2010.

[Pen08a] Karl-Heinz Pennemann. An algorithm for approximating the satisfiability problem of high-level conditions. *Electronic Notes in Theoretical Computer Science*, 213(1):75–94, 2008.

[Pen08b] Karl-Heinz Pennemann. Resolution-Like Theorem Proving for High-Level Conditions. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *LNCS*, pages 289–304, Berlin/Heidelberg, 2008. Springer.

[Pen09] Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, University of Oldenburg, 2009.

[Pos13] Christopher M. Poskitt. *Verification of graph programs*. PhD thesis, University of York, 2013.

[PP10] Christopher M. Poskitt and Detlef Plump. A Hoare Calculus for Graph Programs. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Graph Transformations*, volume 6372 of *LNCS*, pages 139–154, Berlin/Heidelberg, 2010. Springer.

[PP12] Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.

[PP13] Christopher M. Poskitt and Detlef Plump. Verifying total correctness of graph programs. *Electronic Communications of the EASST*, 61, 2013.

[PP14] Christopher M. Poskitt and Detlef Plump. Verifying Monadic Second-Order Properties of Graph Programs. In Holger Giese and Barbara König, editors, *Graph Transformations*, volume 8571 of *LNCS*, pages 33–48, Berlin/Heidelberg, 2014. Springer.

[Rad13] Hendrik Radke. HR* graph conditions between counting monadic second-order and second-order graph formulas. *Electronic Communications of the EASST*, 61, 2013.

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, 157(1):39–59, 2006.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 479–485, Berlin/Heidelberg, 2004. Springer.

[Ren08] Arend Rensink. Explicit State Model Checking for Graph Grammars. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 114–132. Springer, Berlin/Heidelberg, 2008.

[RKL18] Fazle Rabbi, Lars Michael Kristensen, and Yngve Lamo. Analysis and Evaluation of Conformance Preserving Graph Transformation Rules. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Model-Driven Engineering and Software Development*, volume 991 of *CCIS*, pages 284–307, Berlin/Heidelberg, 2018. Springer.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations.* World Scientific, River Edge, 1997.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163, Berlin/Heidelberg, 1994. Springer.

[SK08] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, pages 411–425, Berlin/Heidelberg, 2008. Springer.

[SLO17] Sven Schneider, Leen Lambers, and Fernando Orejas. Symbolic Model Generation for Graph Properties. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, volume 10202 of *LNCS*, pages 226–243, Berlin/Heidelberg, 2017. Springer.

[SLO18] Sven Schneider, Leen Lambers, and Fernando Orejas. Automated reasoning for attributed graph properties. *International Journal on Software Tools for Technology Transfer*, 20(6):705–737, 2018.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144, Berlin/Heidelberg, 2000. Springer.

[Ste15] Dominik Steenken. *Verification of lnfinite-State Graph Transformation Systems via Abstraction.* PhD thesis, University of Paderborn, 2015.

[Stü16] Jan Stückrath. *Verification of Well-Structured Graph Transformation Systems.* PhD thesis, Universität Duisburg-Essen, 2016.

[SV03] Ákos Schmidt and Daniel Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 – The Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, Heidelberg, 2003. Springer.

[SW11] Dominik Steenken and Daniel Wonisch. Using Shape Analysis to Verify Graph Transformations in Model Driven Design. In *9th IEEE International Conference on Industrial Informatics*, pages 457–462. IEEE, 2011.

[SWW11] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Sound and Complete Abstract Graph Transformation. In Adenilso Simao and Carroll Morgan, editors, *Formal Methods: Foundations and Applications*, volume 7021 of *LNCS*, pages 92–107, Berlin/Heidelberg, 2011. Springer.

[Tae00] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In Manfred Nagl, Andreas Schürr, and Manfred Münch, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *LNCS*, pages 481–488, Berlin/Heidelberg, 2000. Springer.

[Won10] Daniel Wonisch. Increasing the preciseness of shape analysis for graph transformation systems. Master's thesis, University of Paderborn, 2010.

[ZR12] Eduardo Zambon and Arend Rensink. Graph Subsumption in Abstract State Space Exploration. In Anton Wijs, Dragan Bosnacki, and Stefan Edelkamp, editors, *Proceedings of First Workshop on Graph Inspection and Traversal Engineering (GRAPHite 2012)*, volume 99 of *EPTCS*, pages 35–49. Open Publishing Association, 2012.

Appendix A.

Additional Examples for Chapter 5: General Approach

This chapter of the appendix contains a number of additional and more detailed examples relating to the general approach described in Chapter 5. Table A.1 provides a short overview.

Example A.1 (Seq-construction, example system)**.** This example is a repetition of Example 5.13 (p. 90) and its computation of $\mathrm{Seq}^g_2(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, with more detailed examples for the construction's individual steps. In particular, we have a graph transformation system $GTS = (TG, \mathcal{R})$, where the set of rules consists only of the graph rule $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ (hence, $\mathcal{R} = \{\mathsf{f2f'}\}$). In addition, we have a safety property $\mathcal{F} = \neg F_1 = \neg\exists i_{P^F_1}$. The rule and the graph constraint $\neg F_1$ are shown in Figures A.1(a) and A.1(b), respectively; they are the same as in Example 5.1 (p. 69).

With respect to the graph rule, we will distinguish between $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ and $\mathsf{f2f'} = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), \mathrm{true}, \mathrm{true}\rangle$. The former will refer to the appearance of rule $\mathsf{f2f'}$ in the context of steps $\mathrm{SC_1}$-1 to $\mathrm{SC_1}$-5 and the latter to its appearance in the context of steps $\mathrm{SC_k}$-1 to $\mathrm{SC_k}$-5, although the rules' contents are identical (i.e. $L_1 = L_2$ and so on).

As explained, we will compute $\mathrm{Seq}^g_2(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, which would be appropriate in order to determine whether $\mathcal{F}$ is a 2-inductive invariant for GTS under a guaranteed constraint with the trivial value true. Here, we will not consider the guaranteed constraint $\mathcal{H}$ introduced in Example 5.1 (p. 69) for reasons of (visual) complexity.

Figure A.2 shows one s/t-pattern sequence (of length 2) seq_2 that is contained in the set $\mathrm{Seq}^g_2(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. The construction and origin of its individual parts are explained in Examples A.2-A.10 below.

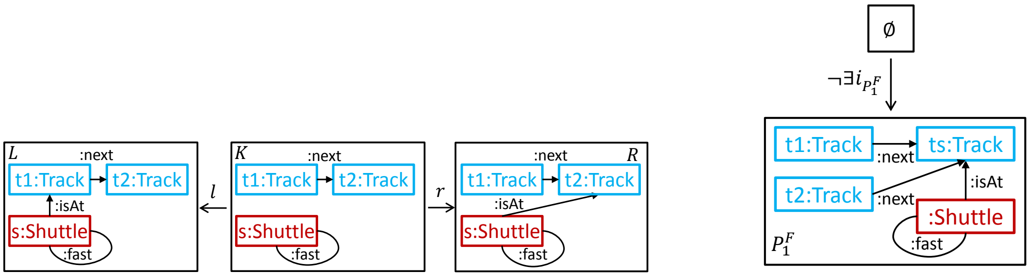
In particular,

$$\begin{aligned} seq_2 = {} & src_1 && \Rightarrow_{\mathsf{f2f'}} && (tar_1, src_2^+) && \Rightarrow_{\mathsf{f2f'}} && tar_2 \\ = {} & \exists(e'_R, ac'_E) \wedge ac_{\mathcal{F}_1} && \Rightarrow_{\mathsf{f2f'}} && (\exists(e_R, ac_E), \exists(e_L, ac_E)) && \Rightarrow_{\mathsf{f2f'}} && \bigvee_{i \in I} \exists t_2^i, \end{aligned}$$

where the steps, their computations, and the corresponding examples and figures for this particular s/t-pattern sequence are listed in Table A.2. Note that some intermediate steps are

Table A.1. – List of examples in Appendix A

Element	Description
Example A.1	Running example and fragments of result of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.2	Step $\mathrm{SC_1}$-1 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.3	Step $\mathrm{SC_1}$-2 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.4	Step $\mathrm{SC_1}$-3 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.5	Steps $\mathrm{SC_1}$-4 and $\mathrm{SC_1}$-5 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.6	Step $\mathrm{SC_k}$-1 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.7	Step $\mathrm{SC_k}$-$1^+$of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.8	Step $\mathrm{SC_k}$-2 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.9	Step $\mathrm{SC_k}$-3 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.10	Steps $\mathrm{SC_k}$-4 and $\mathrm{SC_k}$-5 of $\mathrm{Seq}^g_2(\mathcal{R}, F_1)$
Example A.1	Existence of satisfying transformation sequences (Lemma 5.14)



(a) Graph rule f2f′ **(b)** Safety property $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

Figure A.1. – Graph rule f2f′ and graph constraint $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$

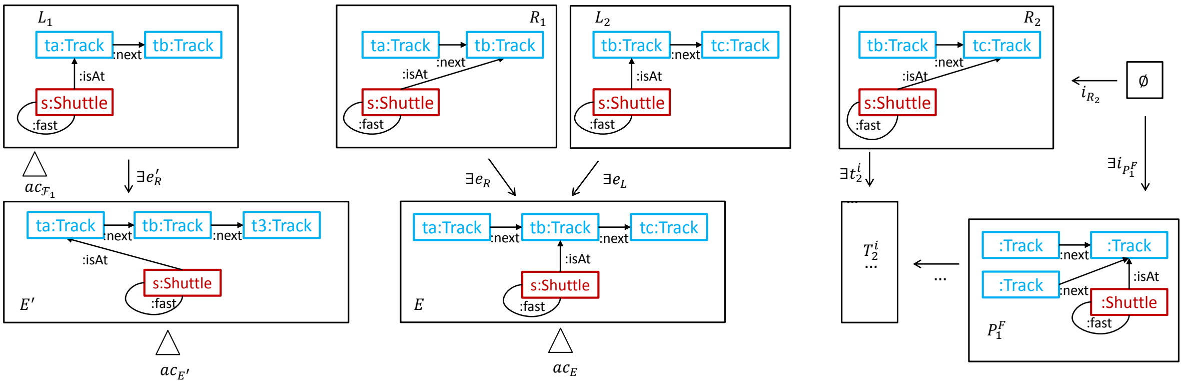


Figure A.2. – The s/t-pattern sequence $seq_2 = src_1 \Rightarrow_{\mathsf{f2f}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$ with $seq_2 \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$

not depicted in the figure. For example, src_2 is not shown because it is part of $src_2^+ = \exists(e_L, ac_E)$ with $ac_E = \mathrm{Shift}(e_L, src_2)$) in the final sequence. Roughly and intuitively, the end result is this: the disjunction over the existential conditions $\exists t_2^i$ describes all possibilities where the application of rule f2f′ has led to a shuttle driving fast on a switch; then, reverse applications of the rule via the L-construction determine the situation before those rule applciations. The sequences in $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ differ by their overlappings of the first and second rule (which are both f2f′). In seq_2, that overlapping is represented by the morphism pair (e_R, e_L) and the graph E. △

Example A.2 (step SC$_1$-1)**.** To construct $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, we first construct $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$.

Table A.2. – Computation steps of $seq_2 \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$

Step	Computation	Figure	Example
SC$_1$-1	$tar_2 = \bigvee_{i\in I} \exists t_2^i = \mathrm{Shift}(i_{R_2}, F_1)$	A.3	A.2
SC$_1$-2	$src'_2 = \bigvee_{i\in I} \exists s_2^i = \mathrm{L}(\mathsf{f2f'}, tar_2)$	A.4	A.3
SC$_1$-3	$src_2 = \bigvee_{i\in I} \exists s_2^i \wedge ac_{\mathcal{F}_2}$ with $ac_{\mathcal{F}_2} = \mathrm{Shift}(i_{L_2}, \mathcal{F})$	A.5	A.4
SC$_1$-4/5	$seq_1 = src_2 \Rightarrow_{\mathsf{f2f'}} tar_2$	A.6	A.5
SC$_k$-1	$tar_1 = \exists(e_R, ac_E)$ with $ac_E = \mathrm{Shift}(e_L, src_2)$	A.8	A.6
SC$_k$-1$^+$	$src_2^+ = \exists(e_L, ac_E)$	–	A.7
SC$_k$-2	$src'_1 = \exists(e'_R, ac_{E'}) = \mathrm{L}(\mathsf{f2f'}, tar_1)$	A.9	A.8
SC$_k$-3	$src_1 = \exists(e'_R, ac_{E'}) \wedge ac_{\mathcal{F}_1}$ with $ac_{\mathcal{F}_1} = \mathrm{Shift}(i_{L_1}, \mathcal{F})$	A.10	A.9
SC$_k$-4/5	$seq_2 = src_1 \Rightarrow_{\mathsf{f2f'}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f'}} tar_2$	A.11	A.10

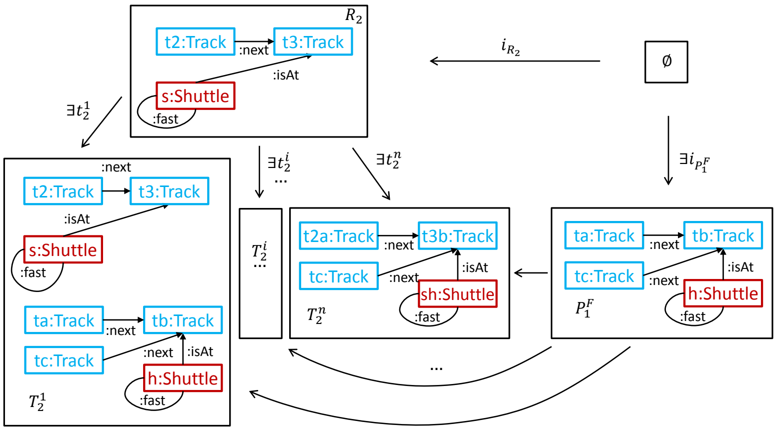


Figure A.3. – Step SC$_1$-1: $tar_2 = \mathrm{Shift}(i_{R_2}, \exists i_{P_1^F}) = \bigvee_{i \in I} \exists t_2^i$

Consider the diagram below for the graphs involved in the construction and Figure A.3 for contents of some of the graphs. Given $\mathcal{R} = \{\mathsf{f2f'}\}$ and with $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true} \rangle$ and $\neg\mathcal{F} = F = \exists i_{P_1^F}$, our first step is the computation of $tar_2 = \mathrm{Shift}(i_{R_2}, \exists i_{P_1^F})$. The result

$$tar_2 = \mathrm{Shift}(i_{R_2}, \exists i_{P_1^F}) = \bigvee_{i \in I} \exists t_2^i$$

is a target pattern over (the right side of) $\mathsf{f2f'}$ and symbolically describes all (disjunctively joined) possibilities of a rule application resulting in the violation of our safety property $\mathcal{F}$ – i.e., in a shuttle driving on a switch in mode fast. In particular (as per the Shift-construction), all those possbilities are possible overlappings between the right side R_2 and the (forbidden) graph C. Since, in this example, we only have one graph rule, there is only one target pattern created in step SC$_1$-1.

$$\begin{tikzcd}
ac_{L_2} \triangleright L_2 & K_2 \arrow[l, hook', "l_2"'] \arrow[rr, hook, "r_2"] & & R_2 \arrow[dl, "t_2^1"'] \arrow[d, "\cdots t_2^i \cdots"] \arrow[dr, "t_2^n"] & & \varnothing \arrow[ll, hook', "i_{R_2}"'] \arrow[d, hook, "i_{P_1^F}"] \\
& & T_2^1 & T_2^i \cdots & T_2^n & P_1^F \arrow[l, hook'] \arrow[ll, bend left, "\cdots"] \arrow[lll, bend left]
\end{tikzcd}$$

Note that besides the two example graphs T_2^1 and T_2^n shown in Figure A.3, there are many more, including some graphs that would be nonsensical given our example system and, particularly, the guaranteed constraint $\mathcal{H}$: Graphs with a shuttle simultaneously being in two different speed modes, graphs with a shuttle positioned at two tracks, and others. Since, for the sake of simplicity, this example only calulates $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ – as opposed to, for instance, $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F} \wedge \mathcal{H}, \mathcal{F} \wedge \mathcal{H})$ – there is no basis for exlusion of these cases in this example construction.

While the target pattern tar_2 is a symbolic representation for a possibly infinite number of situations leading to $\neg\mathcal{F}$ after a rule application, we can also imagine specific graphs: any of the graphs T_2^i could be the result of an application of $\mathsf{f2f'}$ via a comatch $m_2' = t_2^i$ for the respective i; then, we obviously have $m_2' \models \exists t_2^i$ and hence, $m_2' \models tar_2$. Since $\bigvee_{i \in I} \exists t_2^i$ only requires the existence (as opposed to absence) of specific elements, any graph extending one of the graphs T_2^i by arbitrary elements (within the typing restrictions) would also be represented by the target pattern. △

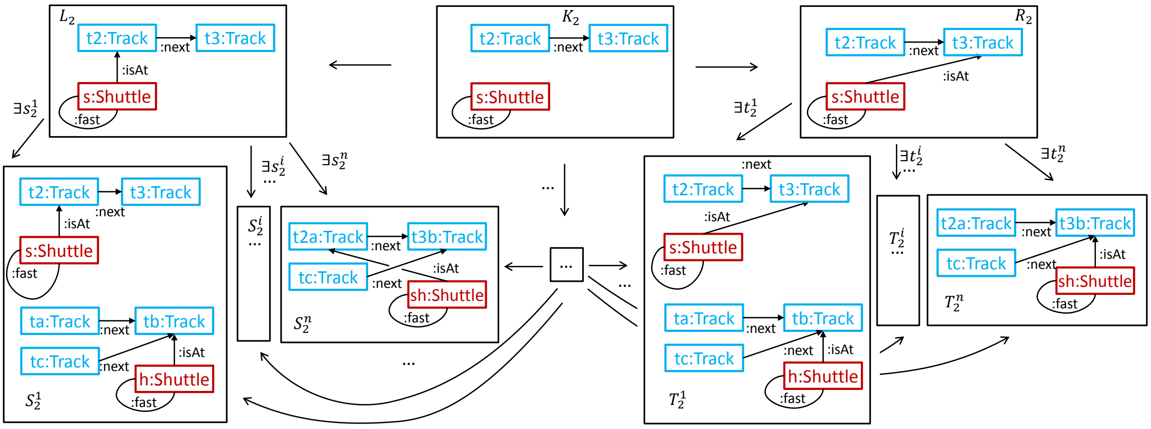


Figure A.4. – Step SC$_1$-2: $src'_2 = \mathrm{L}(\mathsf{f2f}', tar_2) = \bigvee_{i\in I} \exists s_2^i$

Example A.3 (step SC$_1$-2)**.** Given tar_2 (and $\mathsf{f2f}' = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true} \rangle$), our next step is the computation of $src'_2 = \mathrm{L}(\mathsf{f2f}', tar_2)$ as shown in the diagram below and in Figure A.4. As per the L-construction, every condition of the form $\exists t_2^i$ over R_2 is transferred to the left rule side L_2 and results in a condition $\exists s_2^i$. In particular, we have

$$src'_2 = \mathrm{L}(\mathsf{f2f}', tar_2) = \bigvee_{i\in I} \exists s_2^i.$$

$$\begin{tikzcd}
 & ac_{L_2} \triangleright L_2 \arrow[dl, bend right, "s_2^1"'] \arrow[d, "\cdots s_2^i"', "\cdots"] \arrow[dr, bend left, "s_2^n"] & & K_2 \arrow[ll, hook', "l_2"'] \arrow[rr, hook, "r_2"] \arrow[d, "\cdots"] & & R_2 \arrow[dl, bend right, "t_2^1"'] \arrow[d, "\cdots t_2^i"', "\cdots"] \arrow[dr, bend left, "t_2^n"] & \\
S_2^1 & S_2^i\cdots & S_2^n & \cdots \arrow[l] \arrow[ll, bend left, "\cdots"] \arrow[lll, bend left] \arrow[r] \arrow[rr, bend right, "\cdots"'] \arrow[rrr, bend right] & T_2^1 & T_2^i\cdots & T_2^n
\end{tikzcd}$$

In general, the L-construction can also transform existential conditions to false; however, this can only occur if the rule in question creates at least one node, which is not the case here.

The resulting source pattern src'_2 is again a disjunction of existential conditions over the left side of the rule. Figure A.4 shows two example graphs and morphisms $s_2^1 : L_2 \hookrightarrow S_2^1$ and $s_2^n : L_2 \hookrightarrow S_2^n$, with $\exists s_2^i = \mathrm{L}(\mathsf{f2f}, \exists t_2^i)$ being the results of the application of L to the individual existential conditions. Intuitively, the source pattern's individual existential conditions describe situations that lead, after rule application, to the corresponding situations described by the target pattern's individual existential conditions and hence, to a violation of our safety property. △

Example A.4 (step SC$_1$-3)**.** While src'_2 describes possible situations before rule application, the construction does not yet consider whether the rule is actually applicable. Therefore, the Seq-construction conjunctively joins the source pattern src'_2 with the rule's left application condition ac_{L_2} and the rule applicability condition $\mathrm{Appl}(\mathsf{f2f}')$. Here, both conditions are true; since they occur in a conjunction, the source pattern remains unchanged.

Furthermore, this step also takes the additional requirement of the construction of $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ into account: all traversed graphs before the last should satisfy $\mathcal{F}$. Hence, the source pattern is conjunctively combined with the result of the transfer of $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$ over the morphism $i_{L_2} : \varnothing \hookrightarrow L_2$, i.e. with $\mathrm{Shift}(i_{L_2}, \neg\exists i_{P_1^F})$. This is depicted in the diagram below and in Figure A.5.

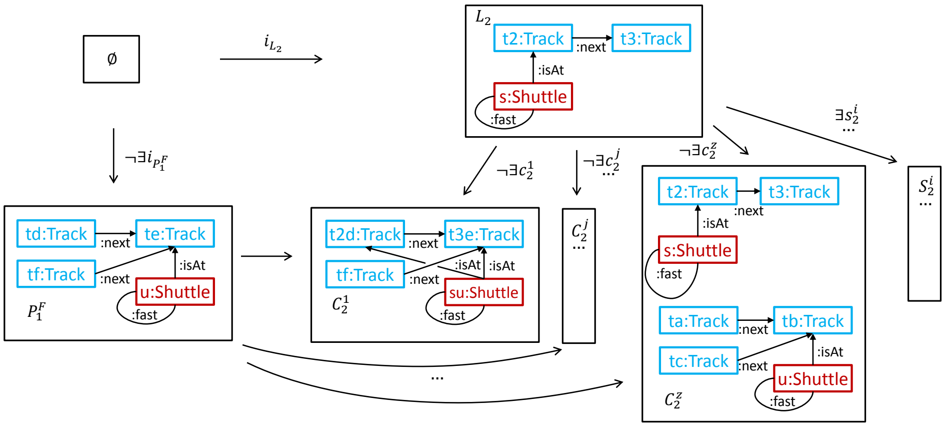


Figure A.5. – Step SC₁-3: $src_2 = src_2' \wedge \text{Shift}(i_{L_2}, \neg F_1)$

$$\begin{tikzcd}
\varnothing \arrow[rr, bend left, "i_{L_2}"] \arrow[d, "i_{P_1^F}"'] & & L_2 \arrow[dl, "c_2^1"'] \arrow[d, "\cdots c_2^j"', "\cdots"] \arrow[dr, "c_2^z"'] \arrow[drr, "s_2^i\cdots"] \arrow[l, phantom, "ac_{L_2} \triangleright"] & & & K_2 \arrow[lll, "l_2"'] \arrow[r, "r_2"] \arrow[d, "\cdots"] & R_2 \arrow[d, "\cdots t_2^i"', "\cdots"] \\
P_1^F \arrow[r] \arrow[rr, bend right, "\cdots"'] \arrow[rrr, bend right] & C_2^1 & C_2^j\cdots & C_2^z & S_2^i\cdots & \cdots \arrow[l, "\cdots"'] \arrow[r, "\cdots"] & T_2^i\cdots
\end{tikzcd}$$

Intuitively, $\text{Shift}(i_{L_2}, \neg\exists i_{P_1^F}) = \bigwedge_{j\in J} \neg\exists c_2^j$ forbids all occurences of P_1^F in the context of L_2 (hence the negated existential conditions). Note that there are again nonsensical graphs (in the context of our example) among the conditions: for example, C_2^1 (Figure A.5) and the respective condition $\neg\exists c_2^1$ forbids a situation involving a shuttle simultaneously existing on two different tracks. As before, the construction does not explicitly exclude such graphs. Furthermore, since all existential conditions $\neg\exists c_2^j$ are conjunctively joined to the source pattern $src_2' = \bigvee_{i\in I} \exists s_2^i$, some of the latter conditions may not be satisfiable at the same time. Although it does not occur in this example, it is possible that a source pattern thusly created and extended is contradictory and equivalent to false.

In summary, this step results in:

$$\begin{aligned}
src_2 &= src_2' \quad \wedge \text{Shift}(i_{L_2}, \neg\exists i_{P_1^F}) \wedge ac_{L_2} \wedge \text{Appl}(\mathsf{f2f'}) \\
&= src_2' \quad \wedge \bigwedge_{j\in J} \neg\exists c_2^j \qquad\quad \wedge \text{true} \wedge \text{true} \\
&= \bigvee_{i\in I} \exists s_2^i \wedge \bigwedge_{j\in J} \neg\exists c_2^j.
\end{aligned}$$

△

Example A.5 (steps SC₁-4 and SC₁-5)**.** Given the results of the previous steps,

$$\begin{aligned}
seq_1 = &\qquad src_2 &&\Rightarrow_{\mathsf{f2f'}} \; tar_2 \\
= &\bigvee_{i\in I} \exists s_2^i \wedge \bigwedge_{j\in J} \neg\exists c_2^j &&\Rightarrow_{\mathsf{f2f'}} \; \bigvee_{i\in I} \exists t_2^i
\end{aligned}$$

is a 1-sequence of source/target patterns (Figure A.6) and, in particular, $seq_1 \in \text{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. Since our example rule set $\mathcal{R} = \{\mathsf{f2f'}\}$ has only one rule, seq_1 is the only s/t-pattern sequence in $\text{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$.

By Theorem T.1g (p. 85), any transformation sequence $trans = G_1 \Rightarrow_{\mathsf{f2f'}} G_2$ satisfying seq_1 will lead to F_1, i.e. $G_2 \vDash F_1$. We can also see that in the target pattern tar_2: any comatch

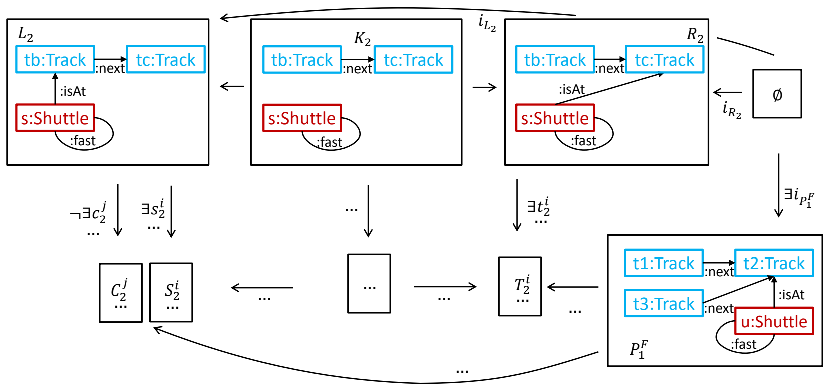


Figure A.6. – Steps SC$_1$-4/5: $seq_1 = src_2 \Rightarrow_{\mathsf{f2f'}} tar_2$ and $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F}) = \{seq_1\}$

$m_2' : R_2 \hookrightarrow G_2$ with $m_2' \vDash tar_2$ must satisfy the disjunction $\bigvee_{i \in I} \exists t_2^i$, which, by construction, describes all possible combinations of R_2 and the forbidden graph P_1^F. △

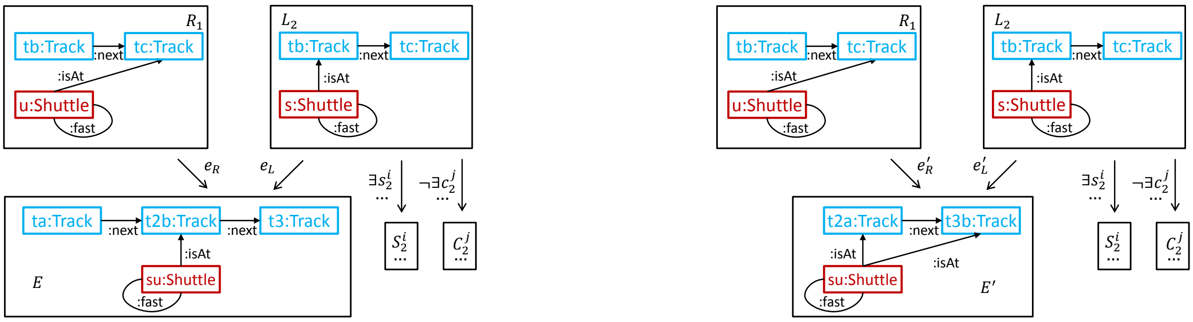
Example A.6 (step SC$_\mathrm{k}$-1)**.** Given the result of $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, we continue the computation of $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ by applying steps SC$_\mathrm{k}$-1 to SC$_\mathrm{k}$-5 of the Seq-construction. Note that we could also relabel the steps as SC$_2$-1 to SC$_2$-5 here.

We consider as input all sequences in $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ and all rules. In particular, we build all combinations of right rule sides of rules in $\mathcal{R}$ and leftmost source patterns – with the respective left rule sides – of s/t-pattern sequences in $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. Here, there is only one rule in $\mathcal{R}$ and only one sequence with one source pattern in $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. In order to better align with the general procedure of Seq, we will denote the rule $\mathsf{f2f'}$ as $\mathsf{f2f'} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true} \rangle$ when appearing in the context of $src_2 \Rightarrow_{\mathsf{f2f'}} tar_2$ (as constructed above) and as $\mathsf{f2f'} = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), \mathrm{true}, \mathrm{true} \rangle$ for the current computation. However, L_1 and L_2 are identical in this example, as are the rules' other components.

We need to consider all pairs of injective and jointly surjective morphisms (e_R, e_L) with $e_R : R_1 \hookrightarrow E$ and $e_L : L_2 \hookrightarrow E$. From each of those pairs, a new s/t-pattern sequence (of length 2) will be constructed. Here, we will focus on only one of these pairs and corresponding sequence, as shown in the diagram below and in Figure A.7(a). There, we see that the respective rule sides partly overlap: the first rule application has moved the shuttle to track t2b, the second will then move it to track t3. Figure A.7(b) depicts another possible morphism pair and corresponding graph E_2.

$$\begin{tikzcd}
ac_{L_1} \triangleright L_1 & K_1 \arrow[l, hook', "l_2"'] \arrow[r, hook, "r_2"] & R_1 \arrow[dr, hook, "e_R"'] & & L_2 \arrow[dl, hook', "e_L"'] \arrow[d, hook, "s_2^i"', "\cdots"] \arrow[dr, hook, "\cdots", "c_2^j"'] & \\
& & & E \arrow[d, hook, "t_1^{ia}"', "\cdots"] \arrow[dr, hook, "t_{1*}^{jd}", "\cdots"'] & S_2^i \cdots \arrow[dl, hook', "\cdots"] & C_2^j \cdots \arrow[dl, hook', "\cdots"] \\
& & & T_1^{ia} \cdots & T_{1*}^{jd} \cdots &
\end{tikzcd}$$

Given the morphisms $e_R : R_1 \hookrightarrow E$ and $e_L : L_2 \hookrightarrow E$ and the graph E, the Seq-construction computes the new target pattern tar_1 over the right rule side R_1. Intuitively, this should create a condition encoding situations where, after one application of $\mathsf{f2f'}$ (because tar_1 is a target pattern over R_1), another application of $\mathsf{f2f'}$ leads to a violation of the safety property (see the previous computation of $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$). Formally, as defined in the first step of the Seq_2^g-construction, $tar_1 = \exists(e_R, ac_E)$ with $ac_E = \mathrm{Shift}(e_L, src_2)$. In particular, we first establish the



(a) Possible morphism pair (e_R, e_L) **(b)** Possible morphism pair (e'_R, e'_L)

Figure A.7. – Two possible injective and jointly surjective morphism pairs for step SC$_\text{k}$-1 of example Seq$_2^g$-construction

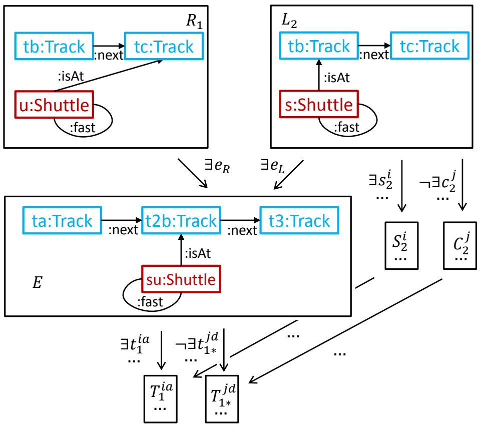


Figure A.8. – Step SC$_\text{k}$-1: $tar_1 = \exists(e_R, \text{Shift}(e_L, src_2))$

context of subsequent rule applications by combining R_1 and L_2 in E via (e_R, e_L) and then shift the source pattern src_2 via e_L to that new context E by computing $ac_E = \text{Shift}(e_L, src_2)$.

Figure A.8 shows the morphism pair and graph E involved in constructing tar_1 but, because of its complexity, does not show specific results of $\text{Shift}(e_L, src_2)$. By the Shift-construction's inductive definition, each existential condition (i.e. each $\exists c_2^j$ and $\exists s_2^i$) again results in a disjunction of several exisential conditions after transformation. Specifically:

$$\begin{aligned}
tar_1 &= \exists(e_R, ac_E)\\
&= \exists(e_R, \text{Shift}(e_L, src_2))\\
&= \exists(e_R, \text{Shift}(e_L, \bigvee_{j\in J} \exists s_2^i \;\wedge \bigwedge_{j\in J} \neg\exists c_2^j))\\
&= \exists(e_R, \bigvee_{j\in J} \text{Shift}(e_L, \exists s_2^i) \wedge \bigwedge_{j\in J} \neg\,\text{Shift}(e_L, \exists c_2^j))\\
&= \exists(e_R, \bigvee_{j\in J} \;\bigvee_{a\in A_j} \exists t_1^{ia} \;\wedge \bigwedge_{j\in J} \;\bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd})
\end{aligned}$$

At this point (if not before), the amount of individual existential conditions and their graphs exceeds what can be seen and understood in reasonable time and effort by a human viewer. Still, the graph E (and its alternatives given other possible morphism pairs (e_R, e_L)) provides an indication of the interaction of subsequently applied rules in the s/t-pattern sequence at hand. △

Example A.7 (step SC$_\text{k}$-1$^+$)**.** Consider tar_1 computed in the previous step and depicted in Figure A.7(a) as one result of combining the source pattern src_1 over L_2 with the right rule side R_1. We extend src_2 to

$$src_2^+ = \exists(e_L, ac_E) \text{ with } ac_E = \text{Shift}(e_L, src_2) \text{ as before.}$$

Then, $(\exists(e_R, ac_E), \exists(e_L, ac_E))$ is a target/source pattern over $(\mathsf{f2f'}, \mathsf{f2f'})$ (and the rule's right and left sides, respectively). It specifies how the first and second rule application are connected. In particular, the shuttles u (right rule side) and s (left rule side) should be mapped to the same shuttle in a satisfying transformation sequence; the same holds for tracks tb (right rule side) and t2 (left rule side) and the respective edges. △

Example A.8 (step SC$_\text{k}$-2)**.** We apply the L-construction to compute the source pattern corresponding to the target pattern tar_1 and application of $\mathsf{f2f'}$ as depicted in the diagram below and in Figure A.9. In particular,

$$\begin{aligned}
src'_1 &= \text{L}(\mathsf{f2f'}, tar_1)\\
&= \text{L}(\mathsf{f2f'}, \exists(e_R, \bigvee_{j\in J}\bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd}))\\
&= \exists(e', \bigvee_{j\in J}\bigvee_{a\in A_j} \text{L}(\mathsf{f2f'}^*, \exists t_1^{ia}) \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\,\text{L}(\mathsf{f2f'}^*, \exists t_{1*}^{jd})) \text{ with } \mathsf{f2f'}^* = (E' \hookleftarrow K'_1 \hookrightarrow E)\\
&= \exists(e', \bigvee_{j\in J}\bigvee_{a\in A_j} \exists s_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists s_{1*}^{jd}).
\end{aligned}$$

$$\begin{tikzcd}
 & ac_{L_1}\triangleright L_1 \arrow[d, "e'"] & K_1 \arrow[l, hook', "l_1"'] \arrow[r, hook, "r_1"] \arrow[d, hook] & R_1 \arrow[dr, hook, "e_R"] & & \\
 & E' \arrow[dl, hook, "s_1^{ia}"'] \arrow[d, hook, "s_{1*}^{jd}"] \arrow[d, phantom, "\cdots"'] & K'_1 \arrow[l, hook'] \arrow[rr, hook] \arrow[d, hook, "\cdots"'] & & E \arrow[d, hook, "t_1^{ia}"'] \arrow[d, phantom, "\cdots"] \arrow[dr, hook, "t_{1*}^{jd}"] & \\
S_1^{ia}\cdots & S_{1*}^{jd}\cdots & \cdots \arrow[l, hook', "\cdots"'] \arrow[rr, hook, "\cdots"] \arrow[ll, bend left, "\cdots"] \arrow[rrr, bend right, "\cdots"'] & & T_1^{ia}\cdots & T_{1*}^{jd}\cdots
\end{tikzcd}$$

While the graphs S_1^{ia} and S_{1*}^{jd}, which correspond to the graphs T_1^{ia} and T_{1*}^{jd}, are not depicted, we can see the effect of the L-construction on the graph E and the condition $\exists(e_R, \dots)$: applying the rule moves the shuttle from track ta to track t2b (see Figure A.9). Its speed mode is unchanged. △

Example A.9 (step SC$_\text{k}$-3)**.** Since an s/t-pattern sequence in $\text{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ needs to guarantee satisfiability of $\mathcal{F}$ in each but the last graph of a satisfying transformation sequence, the construction again transfers the constraint $\mathcal{F}$ to the context of the left rule side. This is shown in the diagram below and in Figure A.10. Applicability of the rule must also be ensured; however, Appl($\mathsf{f2f'}$) and its left application condition ac_{L_1} again have the trivial value true.

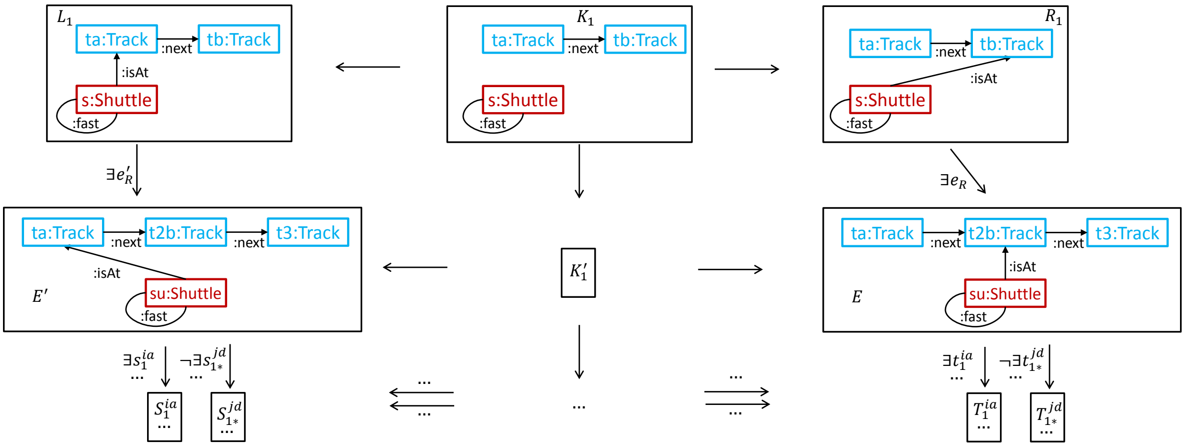


Figure A.9. – Step SC$_\text{k}$-2: $src'_1 = \text{L}(\mathsf{f2f}', tar_1)$

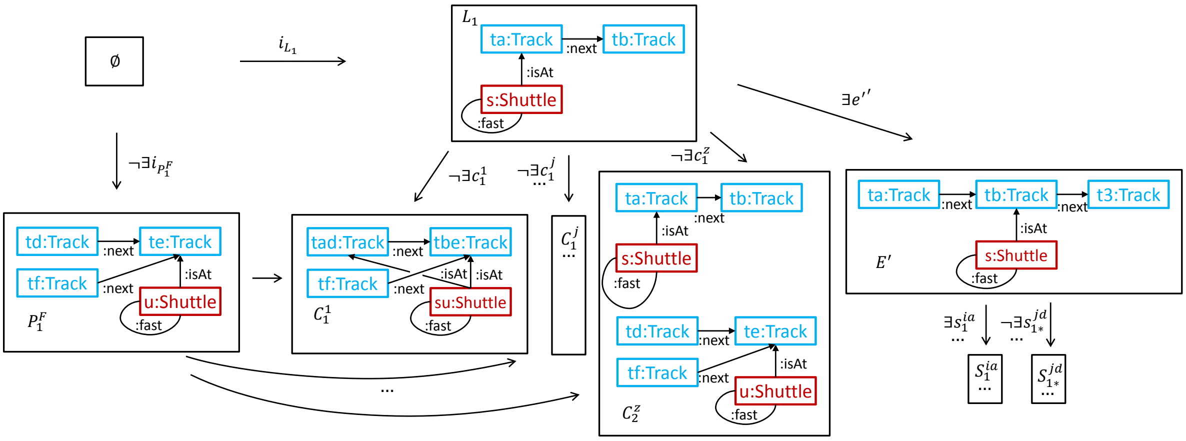


Figure A.10. – Step SC$_\text{k}$-3: $src_1 = src'_1 \wedge \text{Shift}(i_{L_1}, \neg F_1)$

$$\begin{tikzcd}
\emptyset \arrow[rrr, bend left, "i_{L_1}"] \arrow[d, "i_{P_1^F}"] & & & L_1 \arrow[r, phantom, "ac_{L_1} \triangleright"] \arrow[dll, "c_1^1"'] \arrow[dl, "c_1^j"] \arrow[d, "c_1^z"] \arrow[dr, "e'"] & \\
P_1^F \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend right, "\cdots"'] & C_1^1 & C_1^j \cdots & C_1^z & E' \arrow[dl, "s_1^{ia}"'] \arrow[d, "s_{1*}^{jd}"] \\
 & & & S_1^{ia} \cdots & S_{1*}^{jd} \cdots
\end{tikzcd}$$

Thus, we have:

$$\begin{aligned}
src_1 &= \text{Shift}(i_{L_1}, \neg\exists i_{P_1^F}) \wedge src'_1 \wedge \text{Appl}(\mathsf{f2f}') \wedge ac_{L_1} \\
&= \bigwedge_{j\in J} \neg\exists c_1^j \wedge src'_1 \wedge \text{true} \wedge \text{true} \\
&= \bigwedge_{j\in J} \neg\exists c_1^j \wedge src'_1 \\
&= \bigwedge_{j\in J} \neg\exists c_1^j \wedge \exists(e', \bigvee_{j\in J}\bigvee_{a\in A_j} \exists s_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists s_{1*}^{jd})
\end{aligned}$$

Since L_1 and L_2 as left sides of the same rule $\mathsf{f2f}'$ are identical, the morphisms i_{L_1} and i_{L_2} are identical as well. Consequently, $\mathrm{Shift}(i_{L_1}, \neg\exists i_C)$ (as computed here) is equivalent to $\mathrm{Shift}(i_{L_2}, \neg\exists i_C)$ and the graphs C_1^j and C_2^j are isomorphic per index j. (This is also the reason for the duplicate occurence of the index j and index set J). However, note that the original graphs C_2^j and the respective negated existential conditions $\neg\exists c_2^j$ have been transferred multiple times as part of the construction. In src_1, they appear as $\neg\exists s_{1*}^{jd}$ in the context of E'. While the conditions $\neg\exists c_1^j$ (added in the current step) could also be transferred via e' to the context of E', the result would usually not be equivalent to the conjunctively joined conditions $\neg\exists s_{1*}^{jd}$. In more general terms, this is the reason for the Seq-construction to transfer $\mathcal{C}_2$ (here: $\mathcal{F}$) to each source pattern of the s/t-pattern sequence. △

Example A.10 (steps $\mathrm{SC_k}$-4 and $\mathrm{SC_k}$-5)**.** In our running example, we get (among others) a 2-sequence of source/target patterns

$$\mathit{seq}_2 = \mathit{src}_1 \Rightarrow_{\mathsf{f2f}'} (\mathit{tar}_1, \mathit{src}_2^+) \Rightarrow_{\mathsf{f2f}'} \mathit{tar}_2,$$

which is depicted in the diagram below and in Figure A.11. In order to highlight the connections between components and their computational dependencies, we will reiterate that

- $\mathit{src}_1 = \mathrm{L}(\mathsf{f2f}, \mathit{tar}_1) \wedge \mathrm{Shift}(i_{L_1}, \neg F_1)$ (steps $\mathrm{SC_k}$-2 and $\mathrm{SC_k}$-3),
- $(\mathit{tar}_1, \mathit{src}_2^+)$ is a result of transferring src_2 to an overlapping of R_1 and L_2 (steps $\mathrm{SC_k}$-1 and $\mathrm{SC_k}$-1^+),
- $\mathit{src}_2 = \mathrm{L}(\mathsf{f2f}, \mathit{tar}_2) \wedge \mathrm{Shift}(i_{L_2}, \neg F_1)$ is the underlying source pattern to src_2^+ (step $\mathrm{SC_1}$-2),
- and $\mathit{tar}_2 = \mathrm{Shift}(i_{R_2}, F_1)$ ensures that satisfying transformation sequences will lead to $\neg\mathcal{F} = F_1$ (step $\mathrm{SC_1}$-1).

In full detail, we have:

$$\begin{aligned}
\mathit{seq}_2 = {} & \exists(e', \bigvee_{j\in J}\bigvee_{a\in A_j} \exists s_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists s_{1*}^{jd}) \wedge \bigwedge_{j\in J} \neg\exists c_1^j \\
\Rightarrow_{\mathsf{f2f}'} & (\exists(e_R, \bigvee_{j\in J}\bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd}), \\
& \exists(e_L, \bigvee_{j\in J}\bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J}\bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd})) \\
\Rightarrow_{\mathsf{f2f}'} & \qquad \bigvee_{i\in I} \exists t_2^i.
\end{aligned}$$

$$\begin{tikzcd}
\varnothing \arrow[rr, hook, "i_{L_1}"] \arrow[d, hook, "i_{P_1^F}"] & & L_1 \arrow[dl, hook', "c_1^j"', "\cdots"] \arrow[d, hook, "e'"] & K_1 \arrow[l, hook', "l_1"'] \arrow[r, hook, "r_1"] \arrow[d, hook] & R_1 \arrow[dr, hook, "e_R"] & & L_2 \arrow[dl, hook', "e_L"'] \arrow[d, hook, "c_2^j"', "\cdots"] \arrow[dr, hook, "s_2^i", "\cdots"'] & & K_2 \arrow[ll, hook', "l_2"'] \arrow[r, hook, "r_2"] \arrow[d, hook, "\cdots"] & R_2 \arrow[d, hook, "t_2^i"', "\cdots"] & \varnothing \arrow[l, hook', "i_{R_2}"] \arrow[d, hook, "i_{P_1^F}"'] \arrow[llll, hook', bend right, "i_{L_2}"'] \\
P_1^F \arrow[r, hook, "\cdots"] & C_1^j\cdots & E' \arrow[dl, hook', "s_1^{ia}"', "\cdots"] \arrow[d, hook, "s_{1*}^{jd}", "\cdots"'] & K_1'\cdots \arrow[l, hook'] \arrow[rr, hook] \arrow[d, hook, "k"] & & E \arrow[d, hook, "t_1^{ia}"', "\cdots"] \arrow[dr, hook, "t_{1*}^{jd}", "\cdots"'] & C_2^j\cdots \arrow[d, hook, "\cdots"', "\cdots"] & S_2^i\cdots \arrow[dll, hook', dashed] & \cdots \arrow[l, hook', "\cdots"'] \arrow[r, hook, "\cdots"] & T_2^i\cdots & P_1^F \arrow[l, hook', "\cdots"'] \arrow[lll, hook', bend left, "\cdots"] \\
 & S_1^{ia}\cdots & S_{1*}^{jd}\cdots & \cdots \arrow[l, hook', "\cdots"'] \arrow[rr, hook, "\cdots"] \arrow[ll, hook', bend left, "\cdots"] \arrow[rrr, hook, bend right, "\cdots"'] & & T_1^{ia}\cdots & T_{1*}^{jd}\cdots & & & &
\end{tikzcd}$$

Note that, as opposed to $\mathrm{Seq}_1^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, seq_2 is only one of several s/t-pattern sequences in $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. The number of sequences in $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ is equal to the number of injective and jointly surjective morphism pairs (e_R, e_L) and graphs E found in (step $\mathrm{SC_k}$-1).

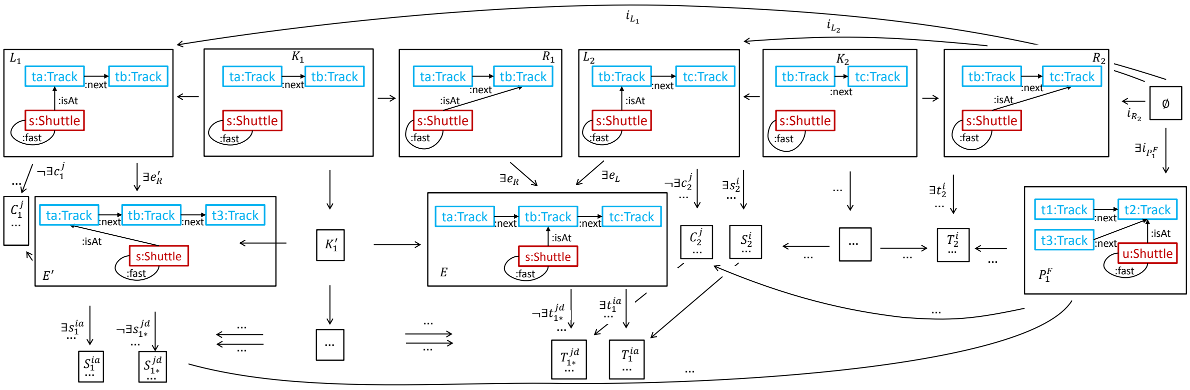


Figure A.11. – Steps SC$_\mathrm{k}$-4/5: $seq_2 = src_1 \Rightarrow_{\mathsf{f2f'}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f'}} tar_2$ with $seq_2 \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$

The first property established by Theorem T.1g (p. 85) guarantees that for any transformation sequence sequence $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} G_1 \Rightarrow_{b_2,m_2,m'_2} G_2$ that leads to $\neg\mathcal{F} = F_1 = \exists i_{P_1^F}$, i.e. to a violation of the safety property, and where that violation does not occur in G_0 or G_1, there is a sequence $seq \in \mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ such that $trans \vDash seq$. This property ensures that all such transformation sequences are represented in $\mathrm{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$. Consequently, we can analyze the constructed s/t-pattern sequences to draw conclusions about all error traces of length 2 that lead to a violation of the safety property.

In particular, seq_2 describes an infinite set of transformation sequences where subsequent shuttle movement in speed mode fast over three subsequently connected tracks leads to the shuttle driving fast on a switch – without this situation occurring earlier in the sequence. Given the second property of Theorem T.1g (p. 85), this is not surprising: given a transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f'},m_1,m'_1} G_1 \Rightarrow_{\mathsf{f2f'},m_2,m'_2} G_2$ with $trans \vDash seq_2$, we know that $trans$ leads to a violation of $\mathcal{F} = \neg F_1$ – with $G_0 \vDash \mathcal{F}$ and $G_1 \vDash \mathcal{F}$.

If we were to compute $\mathrm{Seq}_3^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, we would continue the computation by applying steps SC$_\mathrm{k}$-1 to SC$_\mathrm{k}$-5 (or, specifically, SC$_3$-1 to SC$_3$-5) again, starting with all s/t-pattern sequences of length 2 in $Seq_2(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$, including seq_2. △

This concludes the list of detailed examples for the Seq-construction of the general approach. Following is an example that demonstrates the capability of Lemma 5.14 to deduce from satisfiability of a reduced leftmost source pattern of an s/t-pattern sequence created by the Seq-construction the sequence's satisfiability by a transformation sequence.

Example A.11 (existence of satisfying transformation sequences)**.** Consider the s/t-pattern sequence of length 2 shown in Figure A.11. That sequence is the result of Examples A.1–A.10 for a graph transformation system $GTS = (TG, \{\mathsf{f2f'}\})$ (Figure A.1(a)) and safety property $\mathcal{F} = \neg F_1 = \neg\exists i_{P_1^F}$ (Figure A.1(b)). In order to find a satisfying transformation sequence for seq_2, we attempt to find a graph G_0 that satisfies $src_{1|\varnothing}$ – which is the reduction of src_1 to a graph constraint. In particular, given

$$src_1 = \exists(e', \bigvee_{j\in J} \bigvee_{a\in A_j} \exists s_1^{ia} \wedge \bigwedge_{j\in J} \bigwedge_{d\in D_j} \neg\exists s_{1*}^{jd}) \wedge \bigwedge_{j\in J} \neg\exists c_1^j,$$

we have

$$src_{1|\varnothing} = \exists(i_{E'}, \bigvee_{j\in J} \bigvee_{a\in A_j} \exists s_1^{ia} \wedge \bigwedge_{j\in J} \bigwedge_{d\in D_j} \neg\exists s_{1*}^{jd}) \wedge \bigwedge_{j\in J} \neg\exists i_{C_1^j}.$$

An example graph G_0 satisfying $src_{1|\varnothing}$ is depicted in Figure A.12, with Figures A.12(a) and A.12(b) showing concrete and abstract views of the graphs involved. In particular, we have three subsequent tracks, the last of which is a switch with an additional incoming track. A shuttle, driving in mode fast, is located on the first track.

We can see already why this particular graph (G_0) is a candidate for a first graph of a transformation sequence that leads to the violation of our safety property $\neg F_1$ (Figure 5.12(b)). Two subsequent applications of f2f′ will move the shuttle via track tb to the switch (t3) without changing the speed mode, resulting in a violation. Also, neither G_0 nor the intermediate graph after the first rule application contain the forbidden situation as a subgraph.

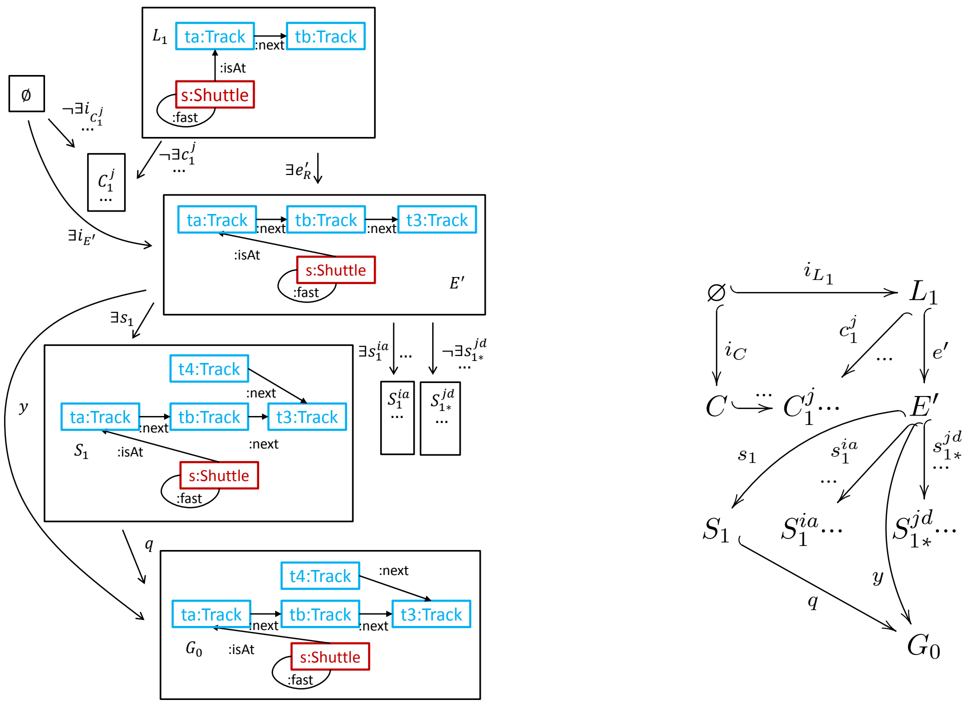
When formally considering whether G_0 satisfies $src_{1|\varnothing}$, we can find an injective morphism $y : E' \hookrightarrow G_0$ such that $y \circ i_{E'} \circ i_{G_0}$. Given that the graphs S_1^{ia} are part of a disjunction, y needs to satisfy $\exists s_1^{ia}$ for specific values of i and a; in other words, we have to find one of the graphs as a subgraph of G_0. Here, such a graph exists and is denoted as S_1 (and also happens to be isomorphic to G_0). The corresponding condition is $\exists s_1$ with the morphism $s_1 : E' \hookrightarrow S_1$. Then, there exists an injective (and bijective) morphism $q : S_1 \hookrightarrow G_0$ such that $q \circ s_1 = y$. Hence, $y \vDash \exists s_1$, which implies $y \vDash \bigvee_{j \in J} \bigvee_{a \in A_j} \exists s_1^{ia}$. Similar to the reasoning about G_0, we can see why S_1 or, more precisely, $\exists s_1$ is a part of the source pattern: it describes one (of several) situations that will lead to a violation of the safety property.

Furthermore, there do not exist specific values for j and d such that there is an injective morphism $x : S_{1*}^{jd} \hookrightarrow G_0$ with $x \circ s_{1*}^{jd} = y$, which then implies $y \vDash \bigwedge_{j \in J} \bigwedge_{d \in D_j} \neg\exists s_{1*}^{jd}$. Finally, a similar argument holds for $i_{G_0} \vDash \bigwedge_{j \in J} \neg\exists i_{C_1^j}$. Note that even without seeing the specific graphs for C_1^j and S_{1*}^{jd}, this makes sense: the second set of graphs and corresponding conditions (C_1^j) come from a transformation of $\neg F_1 = \neg\exists i_{P_1^F}$, which is our safety property, to the context of E'. Since G_0 satisfies the property – while there is a switch (t3), the shuttle is not located on it – it satisfies the corresponding condition fragment in the reduced source pattern. Likewise, S_{1*}^{jd} comes from a similar transformation of the safety property to the context of the intermediate source pattern src_2 (and later src_2^+). Since there is no rule application that can transform G_0 into a graph violating the safety property, such a violation cannot occur in G_0 either.

In summary, we have $i_{G_0} \vDash src_{1|\varnothing}$, implying $G_0 \vDash src_{1|\varnothing}$. Figure A.13(a) shows an alternative graph (G_0') that satisfies $src_{1|\varnothing}$. Figure A.13(b), on the other hand, shows a graph G_0^* that does not satisfy $src_{1|\varnothing}$. While it contains the graph S_1 – three subsequent tracks, the last being a switch with an additional incoming track, and a shuttle (fast) on the leftmost track – we can see that a single application of f2f′ already leads to a violation of the safety property. Hence, a corresponding transformation sequence cannot satisfy our 2-sequence of s/t-patterns seq_2, which was constructed with the requirement that such a violation only happens after two rule applications. In fact, we can find a condition fragment in the conjunction $\bigwedge_{j \in J} \bigwedge_{d \in D_j} \neg\exists s_{1*}^{jd}$ that prevents G_0 from satisfying $src_{1|\varnothing}$; that fragment, here denoted as $\neg\exists s_{1*}$ with $s_{1*} : E' \hookrightarrow S_{1*}$ is shown in Figure A.13(c).

$$\begin{tikzcd}
 & ac_{L_1} \triangleright L_1 \arrow[d, hook, "e'"] \arrow[dddd, hook, bend right, "m_1"'] & & K_1 \arrow[ll, hook', "l_1"'] \arrow[r, hook, "r_1"] \arrow[d, hook] \arrow[ddd, hook, bend right] & R_1 \arrow[rd, hook, "e_R"] \arrow[dddr, hook, bend left, "m_1'"] & & \\
 & E' \arrow[ld, hook, "s_1"'] \arrow[d, hook, "s_1^{ia}\ \cdots"] \arrow[rd, hook, "s_{1*}^{jd}\ \cdots"] \arrow[dd, hook, bend right, "y"] & & K_1' \arrow[ll, hook'] \arrow[rr, hook] \arrow[d, hook, "\cdots"] & & E \arrow[ld, hook, "t_{1*}^{jd}\ \cdots"'] \arrow[d, hook, "t_1^{ia}\ \cdots"] \arrow[rd, hook, "t_1"] \arrow[dd, hook, bend right, "y'"'] & \\
S_1 \arrow[rd, hook, "q"'] & S_1^{ia}\cdots & S_{1*}^{jd}\cdots & \cdots \arrow[l, hook', "\cdots"'] \arrow[ll, hook', bend left, "\cdots"] \arrow[lll, hook', bend left] \arrow[d, hook] \arrow[r, hook, "\cdots"] \arrow[rr, hook, bend right, "\cdots"'] \arrow[rrr, hook, bend right] & T_{1*}^{jd}\cdots & T_1^{ia}\cdots & T_1 \arrow[ld, hook, "q'"] \\
 & G_0 & & D_1 \arrow[ll, hook'] \arrow[rr, hook] & & G_1 &
\end{tikzcd}$$

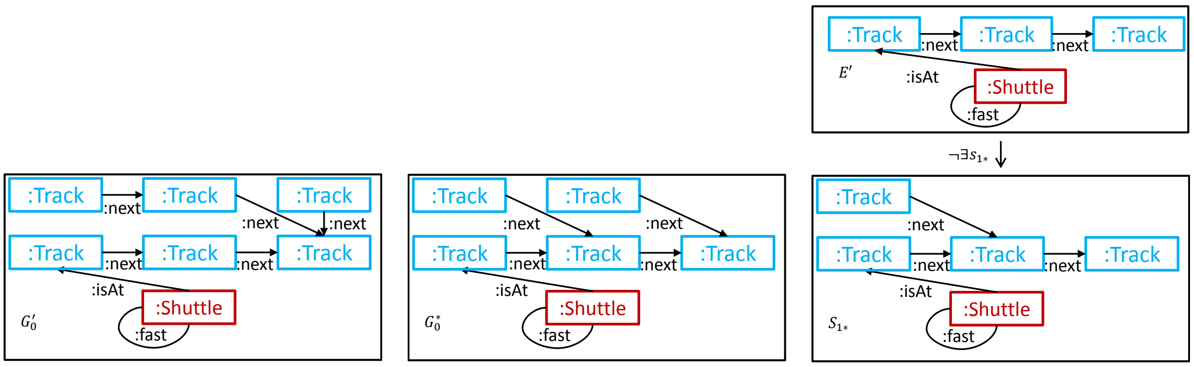
Moving back to G_0, we can find a match $m_1 : L_1 \hookrightarrow G_0$ such that $m_1 \vDash src_1$ and that we can



(a) Concrete view

(b) Abstract view

Figure A.12. – Example graph G_0 with $G_0 \models src_{1|\varnothing}$



(a) G'_0 with $G'_0 \models src_{1|\varnothing}$

(b) G^*_0 with $G^*_0 \not\models src_{1|\varnothing}$

(c) $s_{1*} : E' \hookrightarrow S_{1*}$ as in $\neg\exists s_{1*}$

Figure A.13. – Example graphs and their relation to $src_{1|\varnothing}$

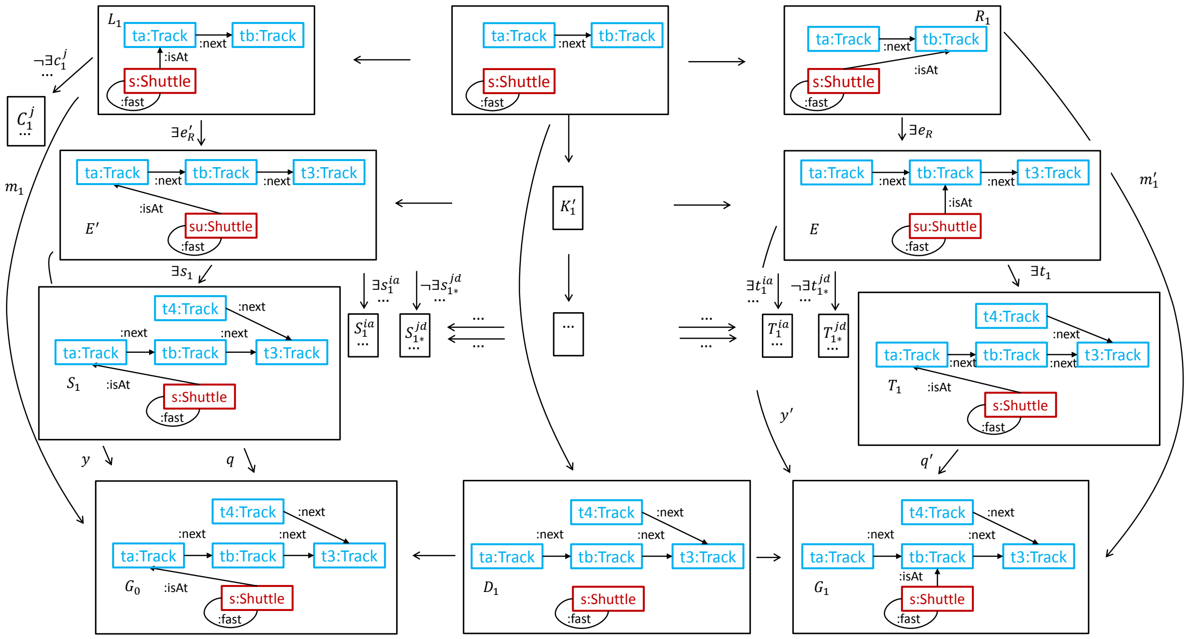


Figure A.14. – Transformation $G_0 \Rightarrow_{\mathsf{f2f}',m_1,m_1'} G_1$ with $m_1 \models src_1$ and $m_1' \models tar_1$

execute a transformation $G_0 \Rightarrow_{\mathsf{f2f}',m_1,m_1'} G_1$ (Figure A.14 and the diagram above).

By the L-construction and L-lemma, we have $m_1' \models tar_1$. Recall that the target pattern tar_1 in our s/t-pattern sequence $seq_2 = src_1 \Rightarrow_{\mathsf{f2f}'} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}'} tar_2$ was computed as

$$tar_1 = \exists(e_R, ac_E) \quad \text{and} \quad ac_E = \bigvee_{j\in J} \bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J} \bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd}.$$

For the purposes of finding a satisfying transformation sequence, the details of the matches and comatches satisfying the source and target patterns are not necessarily relevant. We can compute the graph G_1 by applying the rule $\mathsf{f2f}'$ via m_1 and the two pushouts. However, for the sake of completeness, we will delve deeper into $m_1' \models tar_1$ and the involved conditions and graphs.

In particular, we know that $m_1' \models tar_1$ implies the existence of an injective morphism $y' : E \hookrightarrow G_1$ such that $y' \circ e_R = m_1'$ and that $y' \models \bigvee_{j\in J} \bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J} \bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd}$ (i.e. $y' \models ac_E$). Concerning the first part of the conjunction, we know that $\exists s_1$ in the source pattern was the result of a computation $\mathrm{L}(\mathsf{f2f}', \exists t_1)$, with $t_1 = t_1^{ia}$ and $T_1 = T_1^{ia}$ for some specific values of i and a. Then, $y \models \exists s_1$ is equivalent to $y' \models \exists t_1$ and, as expected, $y' \models \bigvee_{j\in J} \bigvee_{a\in A_j} \exists t_1^{ia}$. While not shown in the diagram, there do not exist specific values for j and d such that any of the $\exists t_{1*}^{jd}$ could be satisfied by y'.

As explained in the proof and depicted in Figure A.15, we can construct an injective morphism $m_2 : L_2 \hookrightarrow G_1$ as $m_2 = y' \circ e_L$; then, $m_2 \models src_2^+$ given that

$$src_2^+ = \exists(e_L, ac_E) \quad \text{with} \quad ac_E = \bigvee_{j\in J} \bigvee_{a\in A_j} \exists t_1^{ia} \wedge \bigwedge_{j\in J} \bigwedge_{d\in D_j} \neg\exists t_{1*}^{jd}$$

as before, because we already know that $y' \models ac_E$. Since (tar_1, src_2^+) is a target/source pattern and m_1' and m_2 satisfy tar_1 and src_2^+ via the morphism y', we have $(m_1', m_2) \models (tar_1, src_2^+)$ (as expected and shown in the proof above).

Furthermore, by construction, we know that

$$ac_E = \mathrm{Shift}(e_L, src_2) \quad \text{and} \quad src_2 = src_2' \wedge \mathrm{Appl}(\mathsf{f2f}') \wedge \mathrm{Shift}(i_{L_2}, \neg\exists i_C),$$

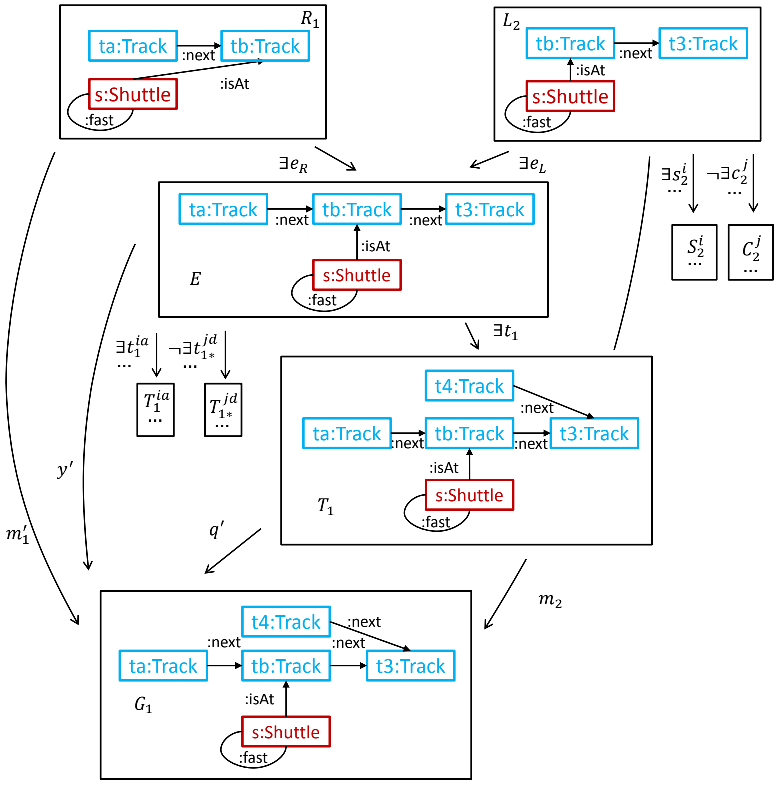


Figure A.15. – Target/source pattern (tar_1, src_2^+) with $(m_1', m_2) \vDash (tar_1, src_2^+)$

implying $m_2 \vDash src_2$, applicability of f2f′ via the match m_2, and the existence of a transformation $G_1 \Rightarrow_{\mathsf{f2f}',m_2,m_2'} G_2$. The transformation and the source and target patterns src_2^+, src_2, and tar_2 are depicted in Figure A.16. In particular, note that the existential condition $\exists t_1$ was the result of shifting a corresponding condition $\exists s_2$ over the morphism e_L as part of computing $ac_E = \text{Shift}(e_L, src_2)$. Hence, we can find an occurrence of S_2 in G_1: formally, $m_2 \vDash \exists s_2$ and thus, $m_2 \vDash src_2$ given that there is no occurrence of any of the negated existential conditions.

Likewise, $\exists s_2$ is the result of transferring $\exists t_2$ over the f2f rule by the L-construction. Finally, $\exists t_2$ is part of the disjunction $\bigvee_{i \in I} t_2^i$. Recall that

$$\bigvee_{i \in I} \exists t_2^i = \text{Shift}(i_{R_2}, \exists i_{P_1^F}) = tar_2$$

is the rightmost target pattern of the s/t-pattern sequence seq_2 under consideration and that $m_2' \vDash tar_2$, seen in the morphism $q'' : T_2 \hookrightarrow G_2$ with $q'' \circ t_2 = m_2'$. Given

$$src_1 \Rightarrow_{\mathsf{f2f}'} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}'} tar_2$$

and with

$$trans = G_0 \Rightarrow_{\mathsf{f2f}',m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f}',m_2,m_2'} G_2,$$

we have $trans \vDash seq_2$, as expceted. The entire transformation sequence $trans$ is depicted in Figure A.17 in compact notation. As shown before, G_2 indeed leads to a violation of our safety property – i.e. $G_2 \vDash \neg\mathcal{F}$ – while no violation ocurrs in G_0 or G_1 – i.e. $G_0 \vDash \mathcal{F}$ and $G_1 \vDash \mathcal{F}$.

Besides showing a satisfying transformation sequence for our example s/t-pattern sequence and suggesting an approach to constructing such transformation sequences, this example serves another purpose: the condition fragments $\exists t_2$, $\exists s_2$, $\exists t_1$ and $\exists s_1$ show how situations exhibiting the desired result (here: a violation of $\mathcal{F}$) are encoded and transformed during the computation of $\text{Seq}_2^g(\mathcal{R}, \neg\mathcal{F}, \mathcal{F})$ and how additional information is accumulated. The graph T_2 is one

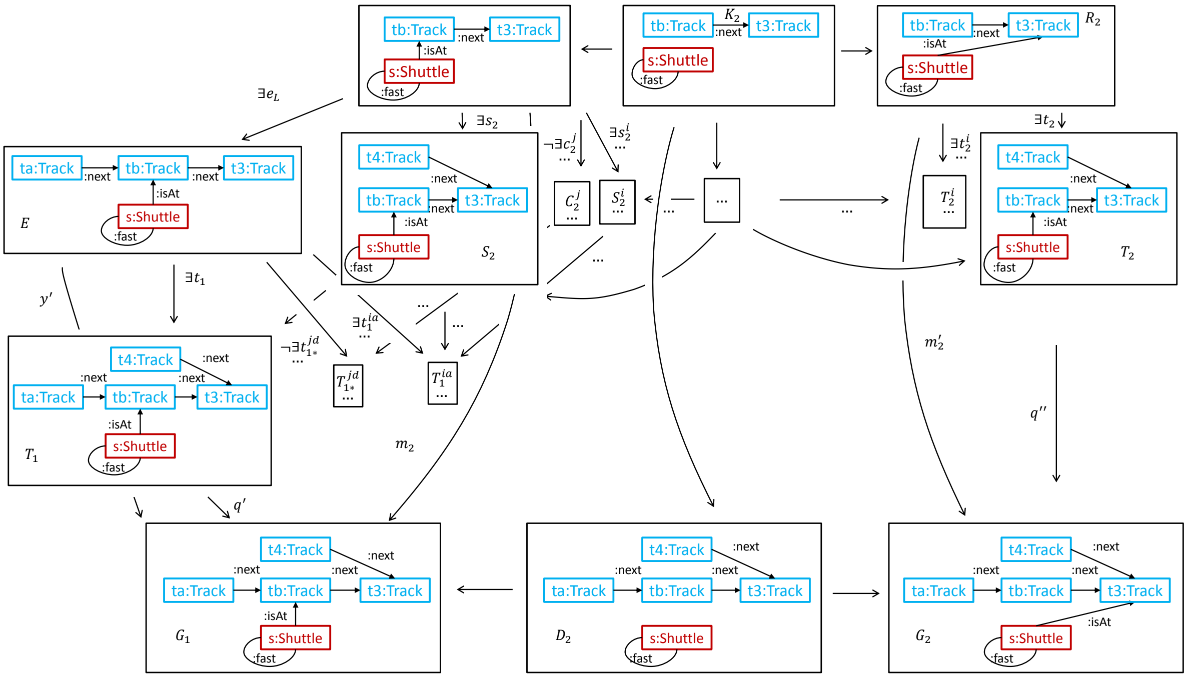


Figure A.16. – Transformation $G_1 \Rightarrow_{\mathsf{f2f}',m_2,m_2'} G_2$ with $m_2 \vDash src_2^+$ and $m_2' \vDash tar_2$

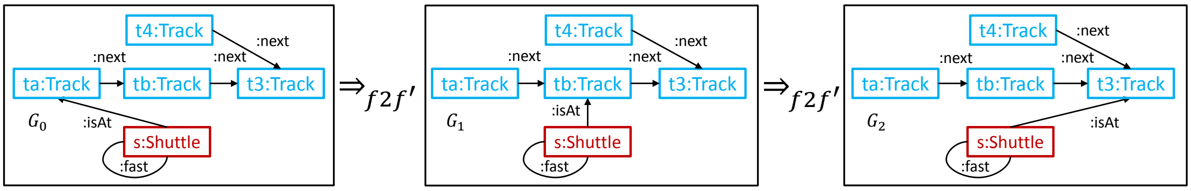


Figure A.17. – Transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f}',m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f}',m_2,m_2'} G_2$ with $trans \vDash seq_2$

of several situations where a rule application has lead to a violation; S_2 then describes the situation before rule application. In the graph T_1 and the corresponding condition $\exists t_1$, that situation has been transferred to the context of the result of a previous application. S_1 then describes the initial situation before the first rule application. Because of that, T_1 encodes more information than T_2; in particular, the graph shows another track (ta) required for a possible previous application of the graph rule $\mathsf{f2f'}$. We have not propagated accumulated information in forward direction. △

Appendix B.

Additional Examples for Chapter 6: Restricted Approach

This chapter of the appendix contains a number of additional and more detailed examples relating to the restricted approach described in Chapter 6. Table B.1 provides a short overview.

Example B.1 (checking implication of graph patterns)**.** Consider two graph patterns $C = \exists(i_P, \neg\exists x_1)$ and $C' = \exists(i_{P'}, \neg\exists x'_1)$ in Figures B.1(a) and B.1(b), respectively. The former expresses the existence of a shuttle in speed mode fast, which is located on a track with a subsequent track ($\exists i'_P$) – with the added requirement that that subsequent track t2 is not a switch, i.e. there is no second track (tA) leading to t2. The latter only describes a shuttle located on a track such that there is no switch (tB) directly ahead.

We can find one and only one injective morphism $m : P \hookrightarrow P'$ (denoted by nodes with the same names). If the question were whether $\exists i'_P$ implied $\exists i_P$, we would have our answer: all graphs satisfying $\exists i'_P$ (i.e. containing P') will satisfy $\exists i_P$, because they will necessarily contain P.

However, both patterns have a (composed) negative application condition each, so that condition (2) of Theorem 6.8 (p. 120) needs to be considered. In particular, we need to compute $\text{Shift}(m, \neg\exists x_1)$, thus shifting the first pattern's negative application condition to the context of the implying pattern (P'). This is partly depicted in Figure B.2. The result, by the Shift-construction, is a composed negative application condition $\bigwedge_{k\in K} \neg\exists x'_{1k}$ over P'; note that only two of the respective graphs X''_{1k} are depicted. Then, we have to find out whether $\neg\exists x'_1$ – the implying pattern's (composed) negative application condition – is strong enough to exclude all of the conditions $\neg\exists x''_{1k}$. This is not the case: while there is an injective morphism $y_1 : X'_1 \hookrightarrow X''_{11}$ such that $y' \circ x'_1 = x''_{11}$, there is no injective morphism $y_8 : X'_1 \hookrightarrow X''_{18}$. Hence (and since there is no other option for a morphism from P to P'), we cannot conclude that C' implies C.

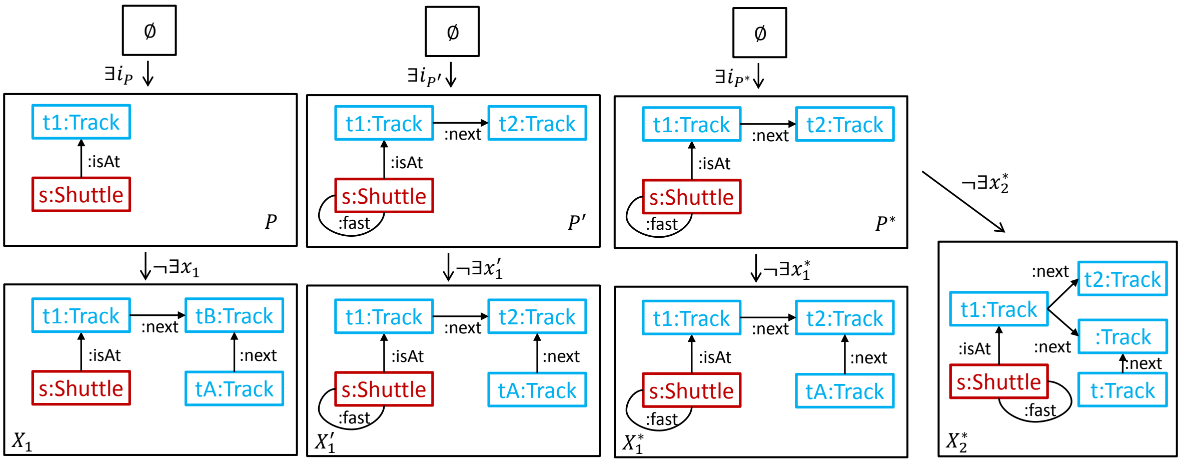
In general, we cannot be sure that C' does not imply C. However, in this particular case, C' does, indeed, not imply C. Knowing that there is no suitable morphism from X'_1 to X''_{18} (and no other negative application condition in C' that we could use) helps us to construct a graph that satisfies C' and does not satisfy C. In particular, X''_{18} is such a graph: it contains P' via x''_{18} and, as we have found, there is no injective morphism $y_8 : X'_1 \hookrightarrow X''_{18}$ such that $y_8 \circ x'_1 = x''_{18}$. Intuitively, we cannot extend x''_{18} such that X''_{18} also contains X'_1. In particular, there are no matching node and edge in X''_{18} for t and its connecting edge because t2 such that x''_{18} is preserved. Thus, X''_{18} satisfies C'. However, while we can find P in X''_{18} (via the morphism $x''_{18} \circ m$), we can also extend that match to X_1 and hence, X''_{18} does not satisfy C.

Consider, on the other hand, C^* (Figure B.1(c)) and C (Figure B.1(a), as before). Since P^* and P are isomorphic, there is a morphism $m : P \hookrightarrow P^*$ as before; the construction $\text{Shift}(m, \neg\exists x_1)$ also has the same result ($\bigwedge_{k\in K} \neg\exists x''_{ik}$) as before. However, with the additional negative application condition $\neg\exists x^*_2$ in the composed negative application condition of P^*, we can now find a morphism $y_8 : X^*_2 \hookrightarrow X''_{18}$ such that $y_8 \circ x^*_2 = x''_{18}$. Indeed, we can find such a morphism (from X^*_1 or X^*_2) for all of the graphs X''_{1k}. Hence, $C^* \vDash C$.

Intuitively, both results make sense. C describes a situation (a shuttle on a track) that forbids the existence of a switch directly ahead of the track. C' describes a more specific situation – an additional track and speed mode fast – which may, at first, suggest implication of C. However, C' only requires t2 not to be a switch (via tA) and does not prohibit the existence of an additional track after t1 that is a switch. That – the absence of any switch directly ahead

Table B.1. – List of examples in Appendix B

Element	Description
Example B.1	Implication check for patterns (Theorem 6.8 (p. 120), Algorithm 6.1)
Example B.2	Running example and fragments of result of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.3	Step SC_1-1 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.4	Step SC_1-2 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.5	Step SC_1-3 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.6	Steps SC_1-4 and SC_1-5 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.7	Step SC_k-1 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.8	Step SC_k-$1^+$of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.9	Step SC_k-2 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.10	Step SC_k-3 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.11	Steps SC_1-4 and SC_1-5 of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$
Example B.12	Steps SC_1-3, SC_1-4, and SC_1-5 with non-trivial left application condition
Example B.13	Other steps with non-trivial left application condition



(a) $C = \exists(i_P, \neg\exists x_1)$ **(b)** $C' = \exists(i_{P'}, \neg\exists x'_1)$ **(c)** $C^* = \exists(i_{P^*}, \neg\exists x^*_1 \wedge \neg\exists x^*_2)$

Figure B.1. – Graph patterns C, C', and C^* with $C' \nvDash C$ and $C^* \vDash C$

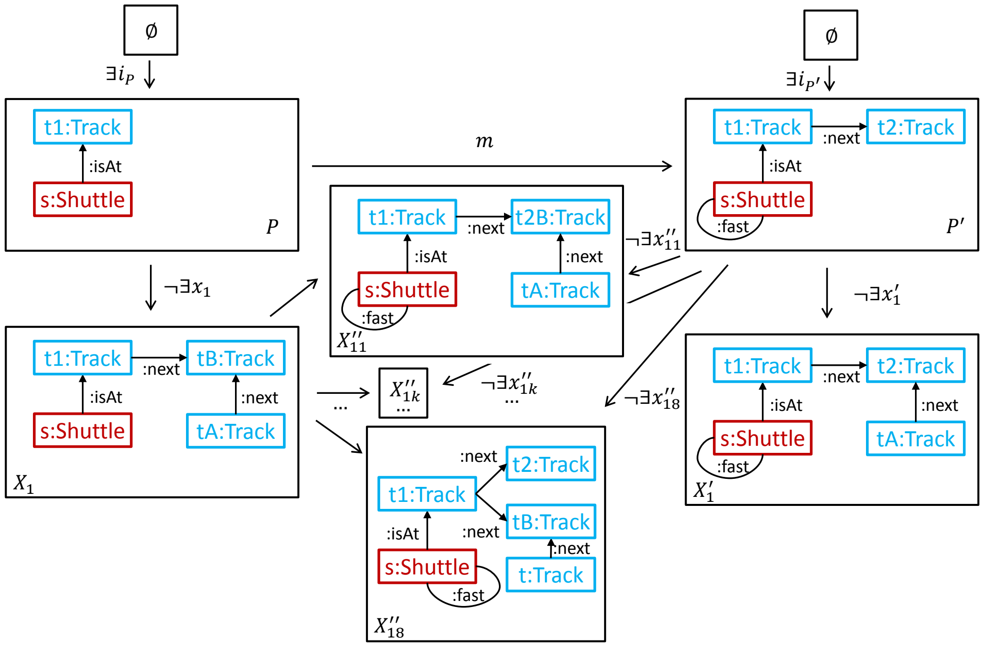
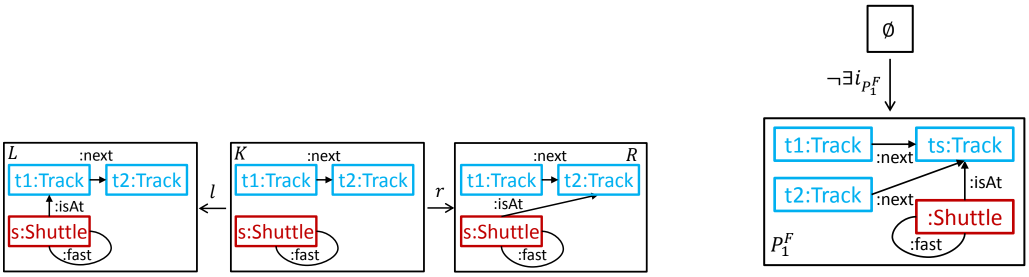


Figure B.2. – Application of Theorem 6.8: injective morphism $m : P \hookrightarrow P'$ and Shift$(m, \neg\exists x_1) = \bigwedge_{k \in K} \neg\exists x''_{1k}$



(a) Graph rule f2f′

(b) Comp. pattern $\mathcal{F} = \neg F_1 = \neg\exists i_{P^F_1}$

Figure B.3. – Graph rule f2f and graph pattern $F = \exists i_P$

– is, however, required by C. C^* then not only forbids t2 to be a switch, but – via $\neg\exists x^*_2$ – prohibits said existence of any switch directly after t1. △

The following examples will serve to illustrate execution of the Seq-construction in more detail.

Example B.2 (Seq-construction, example system)**.** This example is a repetition of Example 6.17 (p. 135) and its computation of $\mathrm{Seq}^r_2(\mathcal{R}, F_1)$, with more detailed examples for the construction's individual steps to follow. We have a graph transformation system with a graph rule f2f′ = $\langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ (hence, $\mathcal{R} = \{\mathsf{f2f'}\}$) and a composed forbidden pattern $\mathcal{F} = \neg F_1 = \neg\exists i_{P^F_1}$. The rule and the graph pattern are shown in Figures B.3(a) and B.3(b), respectively; they are unchanged in comparison to Example 6.1 (p. 111). We will compute $\mathrm{Seq}^r_2(\mathcal{R}, F_1)$, which would be appropriate in order to determine whether $\mathcal{F}$ is a 2-inductive invariant for $GTS = (TG, \mathcal{R})$.

As in the corresponding example for the general approach, we will distinguish between f2f′ = $\langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ and f2f′ = $\langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), \mathrm{true}, \mathrm{true}\rangle$. The former will refer to the appearance of rule f2f in the context of steps SC$_1$-1 to SC$_1$-5 and the latter

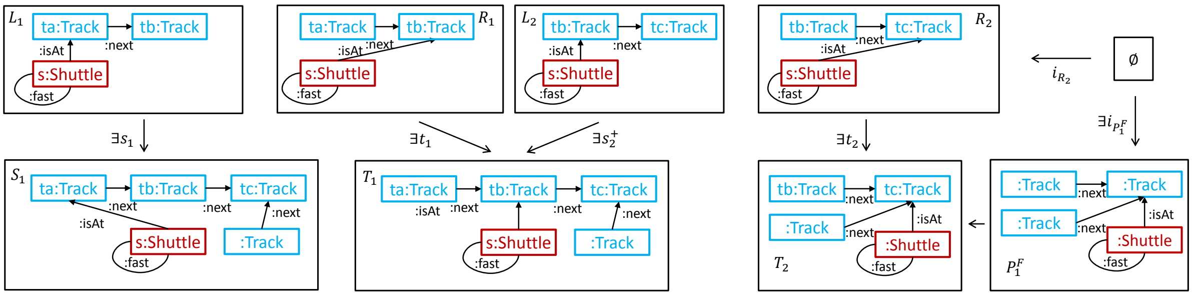


Figure B.4. – An s/t-pattern sequence $seq_2 = src_1 \Rightarrow_{\mathsf{f2f}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$ with $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$

Table B.2. – Computation steps of $\mathrm{Seq}_2^r(\mathcal{R}, F)$

step	computation	Figure	Example
$\mathrm{SC_1}$-1	$tar_2 = \exists t_2$ (and $\bigvee_{j\in J} tar_2^j = \mathrm{Shift}(i_{R_2}, F_1)$)	B.5	B.3
$\mathrm{SC_1}$-2	$src_2' = \exists s_2 = \mathrm{L}(\mathsf{f2f}', tar_2)$	B.6	B.4
$\mathrm{SC_1}$-3	$src_2 = src_2' = \exists s_2$	–	B.5
$\mathrm{SC_1}$-4/5	$seq_1 = src_2 \Rightarrow_{\mathsf{f2f}'} tar_2$	B.7	B.6
$\mathrm{SC_k}$-1	$tar_1 = \exists t_1$	B.9	B.7
$\mathrm{SC_k}$-1^+	$src_2^+ = \exists(s_1' \circ s_2)$	–	B.8
$\mathrm{SC_k}$-2	$src_1' = \exists s_1 = \mathrm{L}(\mathsf{f2f}', tar_1)$	B.10	B.9
$\mathrm{SC_k}$-3	$src_1 = src_1' = \exists s_1$	–	B.10
$\mathrm{SC_k}$-4/5	$seq_2 = src_1 \Rightarrow_{\mathsf{f2f}'} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$	B.11	B.11

to its appearance in the context of steps $\mathrm{SC_k}$-1 to $\mathrm{SC_k}$-5, although the rules' contents are identical. Since the rule does not delete any nodes, it has a trivial applicability condition $\mathrm{Appl}(\mathsf{f2f}') = \mathrm{true}$.

Figure B.4 shows one s/t-pattern sequence (of length 2) seq_2 that is contained in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$. The construction and origin of its individual parts are explained in Examples B.3-B.11 below. In particular,

$$\begin{aligned} seq_2 =& src_1 \Rightarrow_{\mathsf{f2f}'} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}'} tar_2 \\ =& \ \exists s_1 \Rightarrow_{\mathsf{f2f}'} (\exists t_1, \exists s_2^+) \Rightarrow_{\mathsf{f2f}'} \exists t_2, \end{aligned}$$

where the steps, their computations, and the corresponding examples and figures for this particular s/t-pattern sequence are listed in Table B.2. Intuitively, the result can be explained as follows: the existential condition $\exists t_2$ describes one possibility where the application of rule $\mathsf{f2f}'$ has led to a shuttle driving fast on a switch; then, reverse application of the rule via the L-construction determines the situation before that rule application. The resulting situation is described by an existential condition whose context is again combined with the right side of rule $\mathsf{f2f}'$. One such overlapping results in the target pattern $\exists t_2$ – and reverse application of the rule gives us $\exists s_1$, all of which are part of seq_2.

We can see already that the sequence is more specific than its counterpart in Example A.1 (p. A-291) for the general case: the source, target/source, and target patterns $\exists s_1$, $(\exists t_1, \exists s_2^+)$, and $\exists t_2$, are simple existential conditions. Of course, $\mathrm{Seq}_2^r(\mathcal{R}, F)$ will contain more than just that one s/t-pattern sequence seq_2 – and will necessarily contain more sequences than $\mathrm{Seq}_2^g(\mathcal{R}, F_1, \mathrm{true})$. △

Example B.3 (step $\mathrm{SC_1}$-1)**.** To construct $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$, we must first construct $\mathrm{Seq}_1^r(\mathcal{R}, F_1)$. Given $\mathcal{R} = \{\mathsf{f2f}'\}$ and with $\mathsf{f2f}' = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ and $F_1 = \exists i_{P_1^F}$, our first step is

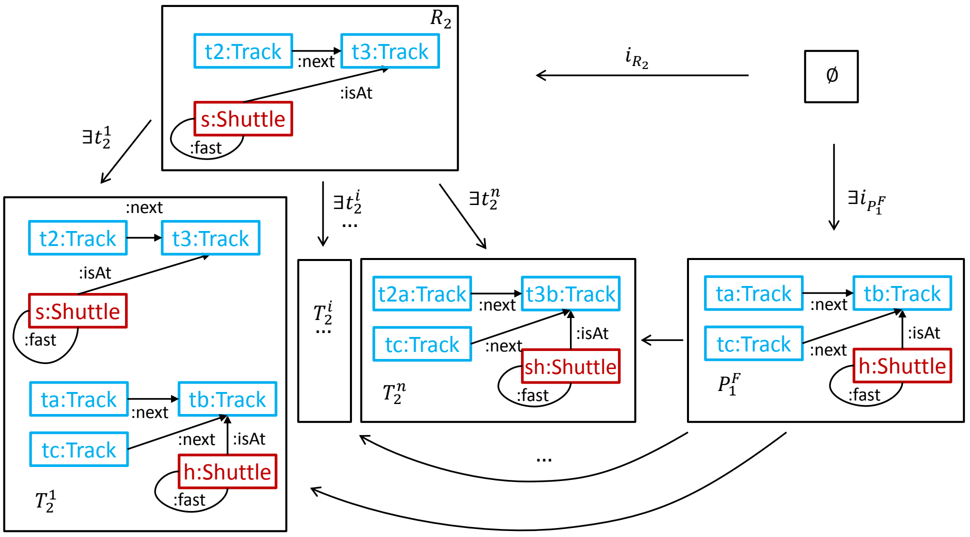


Figure B.5. – Step SC$_1$-1: Shift$(i_R, \exists i_{P_1^F}) = \bigvee_{j\in J} \exists t_2^j$

the computation of Shift(i_{R_2}, F_1) = Shift$(i_{R_2}, \exists i_{P_1^F})$, shown in Figure B.5. The result

$$\text{Shift}(i_{R_2}, \exists i_{P_1^F}) = \bigvee_{j\in J} \exists t_2^j$$

is a disjunction of target patterns over (the right side of) f2f′ and symbolically describes all (disjunctively joined) possibilities of a rule application resulting in the violation of our safety property $\mathcal{F}$ – i.e., in a shuttle driving on a switch in mode fast. In particular (as per the Shift-construction), all those possibilities are possible overlappings between the right side R_2 and the (forbidden) graph P_1^F.

Note that besides the two example graphs T_2^1 and T_2^n shown in Figure B.5, there exist several more, including some graphs that would be nonsensical given our example system and, particularly, the guaranteed constraint $\mathcal{H}$: graphs with a shuttle simultaneously being in two different speed modes, graphs with a shuttle positioned at two tracks, and others. In contrast to our general approach, the Seq-construction in our restricted formal model does not consider an additional constraint – such as a composed guaranteed pattern – to be satisfied by intermediate graphs. While this consideration is integrated into our algorithm, its formal description is only considered as part of the analysis step.

Each of the target patterns tar_2^j is handled individually by the subsequent steps of the Seq-construction and may spawn a new s/t-pattern sequence. In this example, we will focus on only one – the target pattern $tar_2^n = \exists t_2^n$, which we will denote as $tar = \exists t_2$ with $t_2 : R_2 \hookrightarrow T_2$ from here on. This target pattern tar_2 is a symbolic representation for a possibly infinite number of situations leading to F after a rule application (as are the other target patterns not considered here in detail). In particular, tar_2 describes the result of a rule application that has the shuttle moving from one track to another while keeping the speed mode fast and where the target track is a switch. As for specific graphs, consider T_2 itself: it could be the result of an application of f2f′ via a comatch $m_2' = t_2$ for the respective; then, we obviously have $m_2' \models \exists t_2$ and hence, $m_2' \models tar_2$. Any graph extending T_2 by arbitrary elements (within the typing restrictions) would also be represented by the target pattern. △

Example B.4 (step SC$_1$-2). Given $tar_2 = \exists t_2$ (and f2f′ = $\langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \text{true}, \text{true} \rangle$), our next step is the computation of $src_2' = \text{L}(\text{f2f}', tar_2)$ as shown in Figure B.6. Per the L-construction, the condition $\exists t_2$ over R_2 is transferred to the left rule side L_2 and results in a

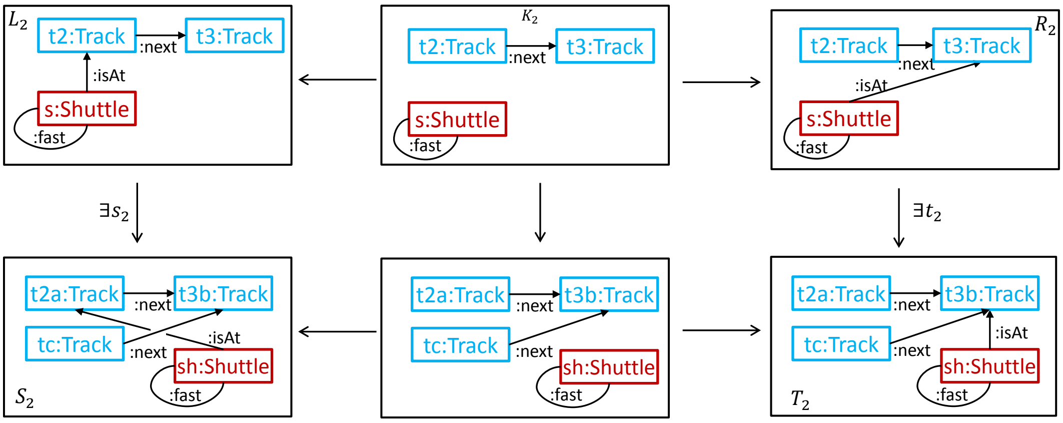


Figure B.6. – Step SC$_1$-2: src'_2 = L(f2f$'$, $\exists t_2$) = $\exists s_2$

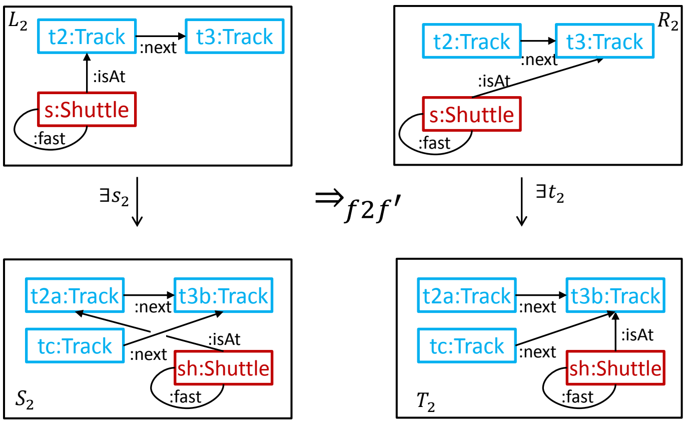


Figure B.7. – Steps SC$_1$-4/5: seq_1 = $src_2 \Rightarrow_{\text{f2f}'} tar_2$ = $\exists s_2 \Rightarrow_{\text{f2f}'} \exists t_2$ with $seq_1 \in \text{Seq}_1^r(\mathcal{R}, F_1)$

condition $\exists s_2$:

$$src'_2 = \text{L}(\text{f2f}', \exists t_2) = \exists s_2.$$

In general, the L-construction can also transform existential conditions to false; however, this would only occur when the pushout complement to $K_2 \hookrightarrow R_2 \hookrightarrow T_2$ would not exist. Since the rule in question does not delete any nodes, this cannot occur in this example, even for other target patterns created in step SC$_1$-1.

Here, the resulting source pattern $src'_2 = \exists s_2$ symbolically describes all situations that, after rule application, lead to any of the situations described by tar_2. Specifically, it describes a shuttle located on a track in speed mode fast to be moved to a subsequent switch, leading to the occurence of our forbidden pattern F. Consider graph S_2: if we choose $s_2 : L_2 \hookrightarrow S_2$ as a match for a rule application, s_2 satisfies src'_2. If we extend S_2 by any number of elements within the typing restrictions – additional tracks, for example – the resulting graphs – or rather, the resulting matches – would also satisfy src'_2. △

Example B.5 (step SC$_1$-3)**.** Since f2f$'$ only has the trivial left application condition true and a trivial applicability condition, this step leaves the source pattern src'_2 unchanged: given ac_{L_2} = true and Appl(f2f$'$) = true, we have Shift(s_2, ac_{L_2}) = true and the conjunction of true with any application condition is equivalent to the application condition itself. Hence, in this example, we have the special case $src_2 = src'_2$. △

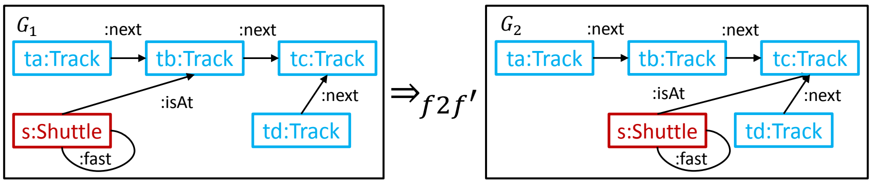


Figure B.8. – Transformation sequence $trans = G_1 \Rightarrow_{\mathsf{f2f}', m_2, m_2'} G_2$ with $trans \vDash seq_1$

Example B.6 (steps SC_1-4 and SC_1-5)**.** Given the results of the previous steps,

$$\begin{aligned} seq_1 &= src_2 \Rightarrow_{\mathsf{f2f}'} tar_2 \\ &= \exists s_2 \Rightarrow_{\mathsf{f2f}'} \exists t_2 \end{aligned}$$

is a 1-sequence of source/target patterns (Figure B.7) and, in particular, $seq_1 \in \mathrm{Seq}_1^r(\mathcal{R}, F_1)$. By construction and because of the structure of $\mathsf{f2f}'$, $\mathrm{Seq}_1^r(\mathcal{R}, F)$ contains a number of sequences equal to the number of operands in the disjunction $\mathrm{Shift}(i_{R_2}, F_1)$.

By Theorem T.1r (p. 130), any transformation sequence $trans = G_1 \Rightarrow_{\mathsf{f2f}'} G_2$ satisfying seq_1 will lead to F_1, i.e. $G_2 \vDash F_1$. We can also see that in the target pattern tar_2: any comatch $m_2' : R_2 \hookrightarrow G_2$ in a satisfying transformation sequence must satisfy tar_2. Then, G_2 must contain T_2 as a subgraph, which exactly describes the forbidden situation (and an embedded right rule side). An example for a satisfying transformation sequence is shown in Figure B.8. △

Example B.7 (step SC_k-1)**.** Given the result of $\mathrm{Seq}_1^r(\mathcal{R}, F_1)$, we continue the computation of $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$ by applying steps SC_k-1 to SC_k-2 of the Seq-construction. (Note that we could also relabel the steps as SC_2-1 to SC_2-5 here).

We consider as input all sequences in $\mathrm{Seq}_1^r(\mathcal{R}, F_1)$ and all rules in $\mathcal{R}$. In particular, we build all overlappings of the right rule side and context graphs of leftmost source patterns of s/t-pattern sequences in $\mathrm{Seq}_1^r(\mathcal{R}, F_1)$. Here, we have only one rule in $\mathcal{R}$, but a couple of 1-sequences of s/t-patterns $\mathrm{Seq}_1^r(\mathcal{R}, F_1)$. In order to better align with the general procedure of the Seq-construction, we will, in this example, denote the rule $\mathsf{f2f}'$ as $\mathsf{f2f}' = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \mathrm{true}, \mathrm{true}\rangle$ in the context of $src_2 \Rightarrow_{\mathsf{f2f}'} tar_2$ (as constructed above) and as $\mathsf{f2f}' = \langle (L_1 \hookleftarrow K_1 \hookrightarrow R_1), \mathrm{true}, \mathrm{true}\rangle$ for the current computation. However, L_1 and L_2 are identical in this example, as are the rules' other components, respectively.

In this example, we will only consider the s/t-pattern sequence $seq_1 = src_2 \Rightarrow_{\mathsf{f2f}'} tar_2 = \exists s_2 \Rightarrow_{\mathsf{f2f}'} \exists t_2$ shown in Example B.6. Then, given $s_2 : L_2 \hookrightarrow S_2$, we need to consider all pairs of injective and jointly surjective morphisms (t_j, s_j') with $t_j : R_1 \hookrightarrow T_j$ and $s_j' : S_2 \hookrightarrow T_j$. From each of those pairs, a new target pattern $\exists t_j$ and, subsequently, a new s/t-pattern sequence (of length 2) will be constructed. For the sake of brevity, we will focus on only one of these morphism pairs and corresponding target pattern in this example. That target pattern, denoted as $tar_1 = \exists t_1$, is shown in Figure B.9.

Intuitively, T_1 encodes a situation where the shuttle, driving in speed mode fast has just moved from track ta to $\mathsf{t2b}$ via the rule $\mathsf{f2f}'$. Furthermore, as described by the sequence $seq_2 = \exists s_2 \Rightarrow_{\mathsf{f2f}'} \exists t_2$, the shuttle is about to move to $\mathsf{t3}$. Since $\mathsf{t3}$ is a switch, this will lead to a violation of our safety property – in particular, it will lead to F_1. Note that the context described in T_1 is more specific when compared to step SC_k-1 in our general approach (Example A.6 (p. A-296)): there, E describes a combination of right and left rule side; here, T_1 is a combination of the right rule side R_1 and the larger context (S_2) the left rule side L_2 appears in. Since neither F_1 nor $\mathsf{f2f}'$ have any non-trivial composed negative application conditions beyond their existential conditions, src_2 and $\exists t_1$ do not have those either.

In the general version of the Seq-construction, this step was (probably) the point where conditions became too complex to understand. In this example, we can still describe and

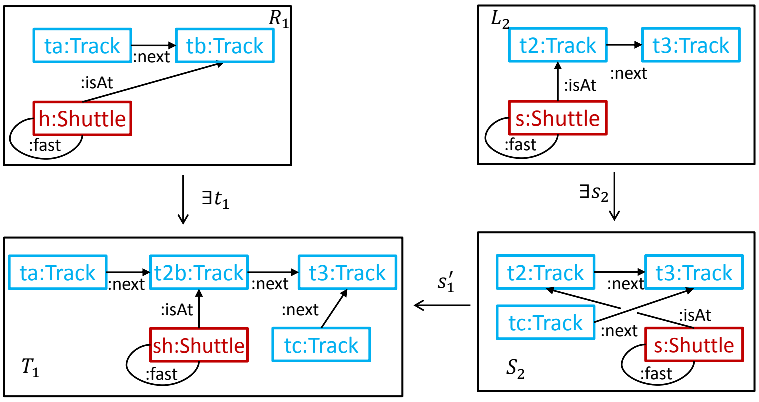


Figure B.9. – Steps SC$_k$-1/1$^+$: morphism pair (t_1, s'_1) (of several such pairs), target pattern $tar_1 = \exists t_1$, source pattern $src_2^+ = \exists s'_1 \circ s_2$, and target/source pattern (tar_1, src_2^+)

understand the context of the rule applications fairly well. This is owed to the lack of a constraint to be fulfilled by a satisfying sequence's intermediate graphs and to the splitting of disjunctions into several target patterns consisting only of singular existential conditions. While the total number of s/t-pattern sequences is actually higher than in the general approach, their complexity has been significantly reduced – and in order to understand the system's behavior and potential violation of our safety property, only one sequence may suffice. △

Example B.8 (step SC$_k$-1$^+$)**.** Consider $tar_1 = \exists t_1$ computed in the previous step and depicted in Figure B.9. This target pattern was created as one of several overlappings between the right rule side R_1 and the source pattern's context S_2. In particular, T_1 extends S_2 by the track ta and its connection to t2b. We therefore extend the source pattern $src_2 = \exists s_2$ to

$$src_2^+ = \exists s'_1 \circ s_2$$

and will denote the composition's result as $s_2^+ = s'_1 \circ s_2$. For different target patterns (not depicted here), the extended source pattern may look different, even if the underlying source pattern used to create the target pattern is the same.

The extension creates a target/source pattern $(tar_1, src_2^+) = (\exists t_1, \exists s'_1 \circ s_2)$ that specifies how the context of the first rule application is connected to the second rule application and source pattern. In particular, (tar_1, src_2^+) shows that the shuttles h (right rule side) and s (left rule side and source pattern) should be mapped to the same shuttle in a satisfying transformation sequence; the same holds for tracks tb (right rule side) and t2 (left rule side and source pattern) and the respective edges. △

Example B.9 (step SC$_k$-2)**.** We apply the L-construction to compute the source pattern corresponding to the target pattern $tar_1 = \exists t_1$ and the application of f2f′ as depicted in Figure B.10. In particular,

$$src'_1 = \mathrm{L}(\mathsf{f2f}', tar_1) = \mathrm{L}(\mathsf{f2f}', \exists t_1) = \exists s_1,$$

which is depicted in Figure B.10.

The result is what we expected: there is a (fast) shuttle located on track ta, which is about to move to track tb, whose subsequent track t3 is a switch. Since $tar = \exists t_1$ was the result of combining a right rule side and the source pattern of a 1-sequence of s/t-patterns, we know that for the resulting sequence of length 2, the shuttle will be moving to track t3, triggering a violation of our safety property. △

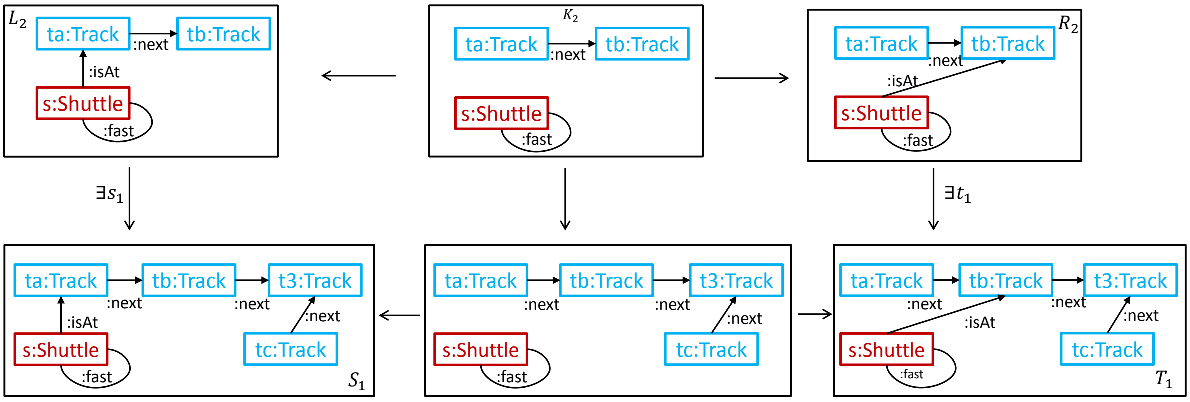


Figure B.10. – Step $\mathrm{SC_k}$-2: $\mathrm{L}(\mathsf{f2f}', \exists t_1) = \exists s_1$

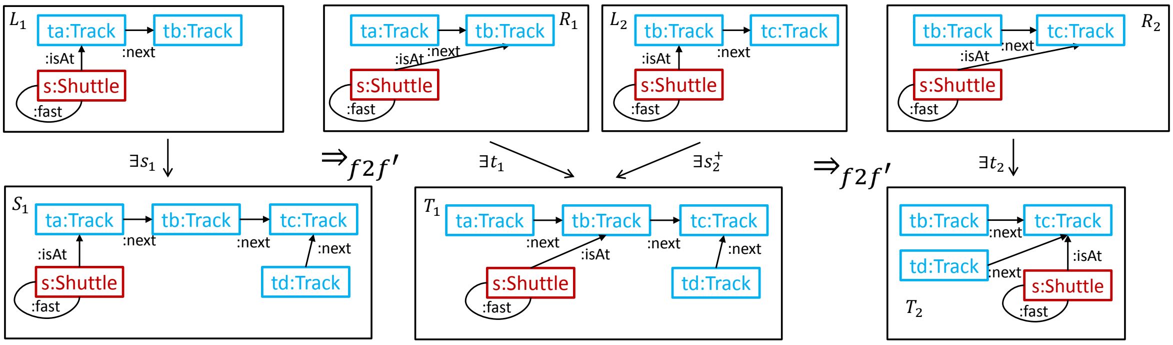


Figure B.11. – Steps $\mathrm{SC_k}$-4/5: $seq_2 = \exists s_1 \Rightarrow_{\mathsf{f2f}'} (\exists t_1, \exists s_2^+) \Rightarrow_{\mathsf{f2f}'} \exists t_2$ with $seq_2 \in \mathrm{Seq}_2^r(\mathcal{R}, F_1)$

Example B.10 (step $\mathrm{SC_k}$-3)**.** As before (step $\mathrm{SC_1}$-3, Example B.5), there is no computation involved in this step for our example: since $\mathsf{f2f}'$ has the trivial left application condition true and a trivial applicability condition, we have $src_1 = src_1' = \exists s_1$. △

Example B.11 (steps $\mathrm{SC_k}$-4 and $\mathrm{SC_k}$-5)**.** In our running example, we get (among others) a 2-sequence of source/target patterns

$$\begin{aligned} seq_2 &= src_1 \Rightarrow_{\mathsf{f2f}'} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}'} tar_2 \\ &= \exists s_1 \Rightarrow_{\mathsf{f2f}'} \ (\exists t_1, \exists s_2^+) \ \Rightarrow_{\mathsf{f2f}'} \exists t_2, \end{aligned}$$

which is depicted in Figure B.11. In order to highlight the connections between components and their computational dependencies, we will reiterate that

- $src_1 = \mathrm{L}(\mathsf{f2f}', tar_1)$ (step $\mathrm{SC_k}$-2),
- tar_1 is a result of combining src_2 and R_1 (step $\mathrm{SC_k}$-1),
- src_2 is the underlying source pattern to src_2^+ and $s_2^+ = s_1' \circ s_2$ (step $\mathrm{SC_k}$-1^+),
- $src_2 = \mathrm{L}(\mathsf{f2f}', tar_2)$ (step $\mathrm{SC_1}$-2),
- and tar_2 is one of the target patterns in the disjunction $\mathrm{Shift}(i_{R_2}, \exists i_{P_1^F}) = \bigvee_{j \in J} \exists t_j$ (step $\mathrm{SC_1}$-1).

The first property established by Theorem T.1r (p. 130) guarantees that for any transformation sequence sequence $trans = G_0 \Rightarrow_{\mathsf{f2f}', m_1, m_1'} G_1 \Rightarrow_{\mathsf{f2f}', m_2, m_2'} G_2$ that leads to $F_1 = \exists i_{P_1^F}$, i.e. a violation of the safety property, there is a sequence $seq \in \mathrm{Seq}_2^r(\mathcal{R}, F)$ such that $trans \models seq$. This property ensures that all such transformation sequences (to $\mathcal{R} = \{\mathsf{f2f}\}$) ending in a shuttle

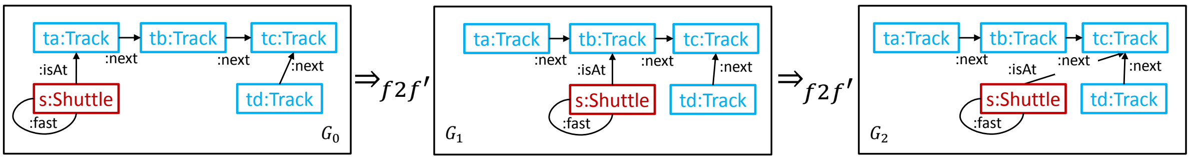


Figure B.12. – Transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f}',m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f}',m_2,m_2'} G_2$; $trans \models seq_2$

in speed mode fast on a switch are represented in $\mathrm{Seq}_2^r(\mathcal{R}, F)$. Consequently, we can analyze the constructed s/t-pattern sequences to draw conclusions about all error traces of length 2 leading to a violation of the safety property.

In particular, seq_2 describes an infinite set of transformation sequences describing the following scenario. A (fast) shuttle is located on a track with a track directly ahead that is followed by a switch. The shuttle moves to the subsequent track twice without changing its speed mode. At the end of the transformation sequence, the shuttle is positioned on a switch in speed mode fast. Given the second property of Theorem T.1r (p. 130), this is not surprising: given a transformation sequence $trans = G_0 \Rightarrow_{\mathsf{f2f}',m_1,m_1'} G_1 \Rightarrow_{\mathsf{f2f}',m_2,m_2'} G_2$ that satisfies seq_2 (i.e., $trans \models seq_2$), we know that $trans$ leads to $F_1 = \exists i_{P_1^F}$. Figure B.12 shows a satisfying transformation sequence in compact notation.

Note the difference in information carried by the rightmost target pattern $tar_2 = \exists t_2$ and the graph G_2 in the transformation sequence. Graphs in transformation sequences are intended to be more precise in describing a situation than target (or source) patterns in s/t-pattern sequences – after all, target patterns are supposed to describe only the minimal context graphs (or matches/comatches, to be precise) have to fulfill. However, the discrepancy here is, technically, not necessary: in fact, the difference lies in the existence of the track td, whose existence is described and required by the target/source pattern (tar_1, src_2^+) and the source pattern src_1. The reason for this lack of information in the rightmost target pattern is our strategy of accumulating information only in backwards direction. This was sufficient in our general approach – the analysis of sequences focused on the leftmost source patterns only. The approach for our restricted formal model, however, analyzes intermediate source and target patterns also. Hence, it is beneficial to also propagate information in forward direction, which is discussed as an extension in Section 7.1. This lack of information does not only concern the patterns' existential conditions, but also their composed negative application conditions, if non-trivial: information about such conditions in src_1 is not available in tar_2.

If we were to compute $\mathrm{Seq}_3^r(\mathcal{R}, F_1)$, we would continue the computation by applying steps SC_k-1 to SC_k-5 (or, specifically, SC_3-1 to SC_3-5) again, starting with all s/t-pattern sequences of length 2 in $Seq_2(\mathcal{R}, F_1)$, including seq_2. △

Example B.12 (steps SC_1-3, SC_1-4, and SC_1-5 with a non-trivial left application condition)**.** In order to illustrate the consequences of a non-trivial left application condition in rules, consider the graph rule $\mathsf{f2f} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), ac_{L_2}, \mathrm{true}\rangle$ with $ac_{L_2} = \neg\exists x_1$ introduced in Example 6.1 (p. 111) and depicted again in Figure B.13. With the additional negative application condition, the rule is only applicable if the track the shuttle is supposed to move to does not have a switch on its subsequent track. However, the target track itself may be a switch: rules check for switches two tracks ahead, not one.

Table B.3 shows the individual computation steps. Since the rest of the rule and the pattern F_1 remain unchanged, some steps of the construction are exeucted as before. After step SC_1-2, we have $src_2' = \exists s_2 = \exists(s_2, \mathrm{true})$ as in Example B.4 (p. B-313). Then, given f2f, we get a source

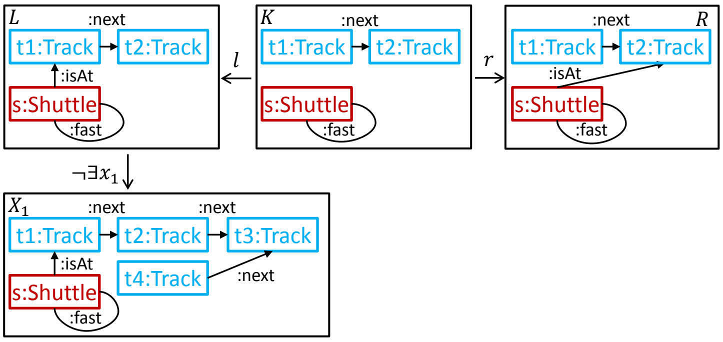


Figure B.13. – Graph rule $\mathsf{f2f} = \langle (L_2 \hookleftarrow K_2 \hookrightarrow R_2), \neg\exists x_1, \text{true}\rangle$

Table B.3. – Computation steps of $\text{Seq}_2^r(\mathcal{R}, F_1)$

step	computation	Figure	Example
SC_1-1	$tar_2 = \exists t_2$ (and $\bigvee_{j\in J} tar_2^j = \text{Shift}(i_{R_2}, F_1)$)	B.5	B.3
SC_1-2	$src'_2 = \exists s_2 = \text{L}(\mathsf{f2f}, tar_2)$	B.6	B.4
SC_1-3	$src_2 = \exists(s_2, ac_{S_2}) = \exists(s_2, \text{Shift}(s_2, ac_{L_2}))$	B.14	B.12
SC_1-4/5	$seq_1 = src_2 \Rightarrow_{\mathsf{f2f}} tar_2$	–	B.12
SC_k-1	$tar_1 = \exists(t_1, ac_{T_1}) = \exists(t_1, \text{Shift}(s'_1, ac_{S_2}))$	–	B.13
$\text{SC}_\text{k}\text{-1}^+$	$src_2^+ = \exists(s'_1 \circ s_2, ac_{T_1})$	–	B.13
SC_k-2	$src'_1 = \exists(s_1, ac'_{S_1}) = \text{L}(\mathsf{f2f}, tar_1)$	–	B.13
SC_k-3	$src_1 = \exists(s_1, ac_{S_1}) = \exists(s_1, ac'_{S_1} \wedge \text{Shift}(s_1, ac_{L_1}))$	B.15	B.13
SC_k-4/5	$seq_2 = src_1 \Rightarrow_{\mathsf{f2f'}} (tar_1, src_2^+) \Rightarrow_{\mathsf{f2f}} tar_2$	B.16	B.13

pattern

$$\begin{aligned} src_2 &= \exists(s_2, \text{true} \wedge \text{Shift}(s_2, ac_{L_2})) \\ &= \exists(s_2, \text{Shift}(s_2, \neg\exists x_1)) \\ &= \exists(s_2, \neg\exists x_2^1 \wedge \neg\exists x_2^2 \wedge \neg\exists x_2^3), \end{aligned}$$

which is depicted in Figure B.14. Specifically, the source pattern's existential condition describes the situation as before: a (fast) shuttle is about to move to a switch. The rule's applicability condition is trivially true and does not affect the source pattern. However, the rule's non-trivial negative application condition states that the rule may only be applied if the target track's subsequent track is not a switch. In the context of the left rule side, this can be expressed by only one negated existential condition. After transfer to the context of the source pattern, the condition is more involved, because the source pattern specifies additional elements (here: the track tc and its connection) beyond the right rule side. There a three negative application conditions, joined conjunctively to a composed negative application condition:

$\neg\exists x_2^1$: if t3b has tc as both its previous and subsequent track (although our guaranteed patterns would not allow that) and if there is an additional track prior to tc, then tc would be a switch one track ahead of t3b, which would prevent application of the rule.

$\neg\exists x_2^2$: if t3b has a subsequent track that has both t3b and tc as its previous tracks (again, this would be forbidden), the unnamed track would be a switch one track ahead of t3b, which would prevent application of the rule.

$\neg\exists x_2^3$: if there is one track ahead of t3b and if that track has an additional track before it –

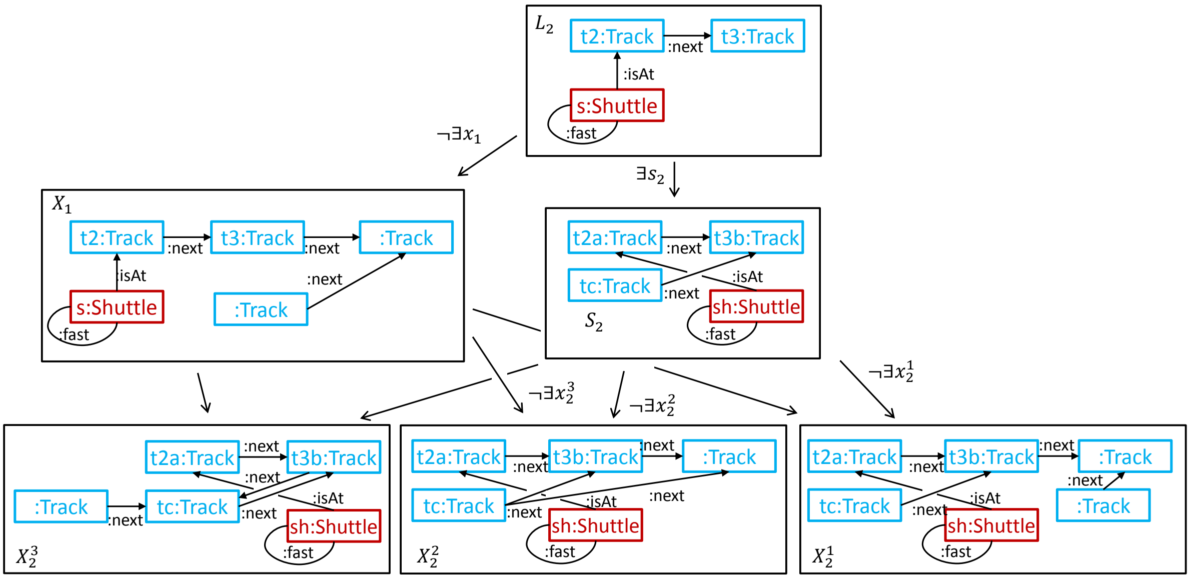


Figure B.14. – Step SC$_1$-3: $src_2 = \exists(s_2, \text{Shift}(s_2, ac_{L_2})) = \exists(s_2, \neg\exists x_2^1 \wedge \neg\exists x_2^2 \wedge \neg\exists x_2^3)$

i.e., if it is a switch – the former track would be a switch one track ahead of t3b, which would prevent application of the rule.

While shifting the left application condition to the context of the source pattern adds complexity to the computation and result, it also provides more information that can be used in the analysis later on. Further, it could even simplify the result: if one of the morphisms x_2^i were an isomorphism, we could discard the source pattern as unsatisfiable. However, that is not the case here.

In steps SC$_1$-4 and SC$_1$-5, we get

$$\begin{aligned} seq_1 &= \qquad src_2 \qquad \Rightarrow_{\mathsf{f2f}} tar_2 \\ &= \exists(s_2, \neg\exists x_2^1 \wedge \neg\exists x_2^2 \wedge \neg\exists x_2^3) \Rightarrow_{\mathsf{f2f}} \exists t_2. \end{aligned}$$

The number of s/t-pattern sequences in $\text{Seq}_1^r(\mathcal{R}, F_1)$ is equal to the number in Example B.6 (p. B-314). △

Example B.13 (other steps with a non-trivial negative application condition)**.** Since src_2 has a non-trivial composed negative application condition, steps SC$_\text{k}$-1–SC$_\text{k}$-5 and their results are different from the previous examples. In particular, $ac_{S_2} = \neg\exists x_2^1 \wedge \neg\exists x_2^2 \wedge \neg\exists x_2^3$ will be part of all new target patterns created in step SC$_\text{k}$-1: for each morphism pair $(t_j : R_1 \hookrightarrow T_j, s'_j : S_2 \hookrightarrow T_j)$, we get a target pattern $\exists(t_j\,\text{Shift}(s'_j, ac_{S_2}))$. Given the morphism pair (t_1, s'_1) used in Example B.7 (p. B-315), we have

$$\begin{aligned} tar_1 &= \exists(t_1, \qquad ac_{T_1}) \\ &= \exists(t_1, \text{Shift}(s'_1, \qquad ac_{S_2})) \\ &= \exists(t_1, \text{Shift}(s'_1, \neg\exists x_2^1 \wedge \qquad \neg\exists x_2^2 \wedge \qquad \neg\exists x_2^3) \\ &= \exists(t_1, \qquad \text{Shift}(s'_1, \neg\exists x_2^1) \wedge \text{Shift}(s'_1, \neg\exists x_2^2) \wedge \text{Shift}(s'_1, \neg\exists x_2^3)). \end{aligned}$$

Unfortunately, even with concrete graphs for the transferred negative application conditions, their meaning may not be clear at first glance due to their number – which is why we leave out details here. However, the (positive) context of a target pattern (T_1 or $\exists t_1$) is usually more important and gives a clear indication about the situations described by the s/t-pattern sequence. Also, unless the composed negative applicadtion condition is contradictory, we can always imagine a satisfying comatch: $t_1 : R_1 \hookrightarrow T_1$ is both a potential comatch and satisfies tar_1; hence, T_1 describes a situation represented by the target pattern.

Then, for step SC$_\text{k}$-1$^+$, we get

$$src_2^+ = \exists(s_1' \circ s_2, ac_{T_1}) \text{ and } (tar_1, src_2^+) = (\exists(t_1, ac_{T_1}), \exists(s_1' \circ s_2, ac_{T_1})),$$

followed by

$$\begin{aligned} src_1' &= \mathrm{L}(\mathsf{f2f}, tar_1) \\ &= \mathrm{L}(\mathsf{f2f}, \exists(t_1, ac_{T_1})) \\ &= \mathrm{L}(\mathsf{f2f}, \exists(s_1, ac_{S_1}')) \end{aligned}$$

in step SC$_\text{k}$-2. The source pattern's existential condition remains unchanged in comparison to Example B.9 (p. B-316) – only its composed negative appliction condition changes.

In step SC$_\text{k}$-3, we again need to shift the rule's negative application condition to the source pattern's context, similar to step SC$_1$-3. We get

$$\begin{aligned} src_1 &= \exists(s_1, \mathrm{Shift}(s_1, ac_{L_1}) \wedge ac_{S_1}') \\ &= \exists(s_1, \mathrm{Shift}(s_1, \neg\exists x_1) \wedge ac_{S_1}') \\ &= \exists(s_1, \neg\exists x_1 \wedge \bigwedge_{u \in U} \neg\exists x_1^u \wedge ac_{S_1}') \\ &= \exists(s_1, ac_{S_1}), \end{aligned}$$

which is depicted in Figure B.15. Note that we have singled out the negative application condition $\neg\exists x_1^0$ because x_1^0 is an isomorphism. Hence, no morphism can satisfy $\exists(s_1, \neg\exists s_1^0)$, which makes src_1 equivalent to false. As before, the rule has a trivial applicability condition.

The resulting s/t-pattern sequence

$$\begin{aligned} seq_2 &= src_1 && \Rightarrow_{\mathsf{f2f}} (tar_1, src_2^+) && \Rightarrow_{\mathsf{f2f}} tar_2 \\ &= \exists(s_1, ac_{S_1}) && \Rightarrow_{\mathsf{f2f}} (\exists(t_1, ac_{T_1}), \exists(s_2^+, ac_{T_1})) && \Rightarrow_{\mathsf{f2f}} \exists t_2 \text{ where} \\ ac_{S_1} &= \neg\exists x_1^0 \wedge \bigwedge_{u \in U} \neg\exists x_1^u \wedge ac_{S_1}' \end{aligned}$$

is depicted in Figure B.16. However, given $src_1 \equiv$ false, no satisfying transformation sequence can exist. In particular, consider $trans = G_0 \Rightarrow_{\mathsf{f2f}', m_1, m_1'} G_1 \Rightarrow_{\mathsf{f2f}', m_2, m_2'} G_2$ used in Example B.11, Figure B.12 (p. B-318). Because of the negative application condition of f2f, $trans$ is not a valid transformation sequence when we replace f2f′ by f2f. In other words, the interaction of two rules f2f in the manner described by seq_2 cannot lead to the combination of right rule side and forbidden pattern used to create the rightmost target pattern in step SC$_1$-1. If this were true for all s/t-pattern sequences in $\mathrm{Seq}_2^r(\mathcal{R}, F_1)$ (using f2f), there would be no transformation sequences that could lead to a violation of the safety property. △

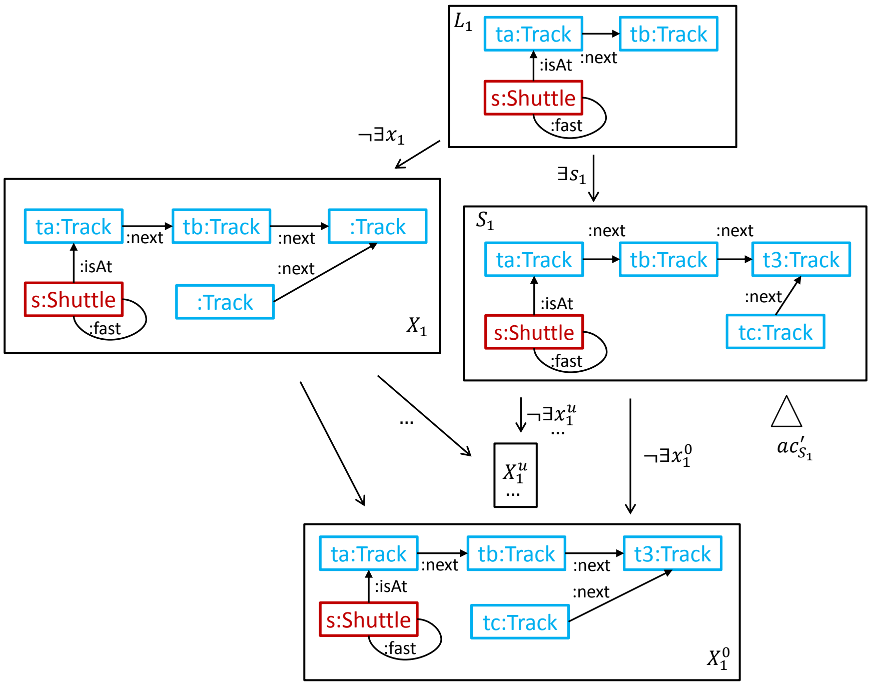


Figure B.15. – Step SC$_{\text{k}}$-3: $src_1 = \exists(s_1, \text{Shift}(s_1, ac_{L_1}) \wedge ac'_{S_1})$

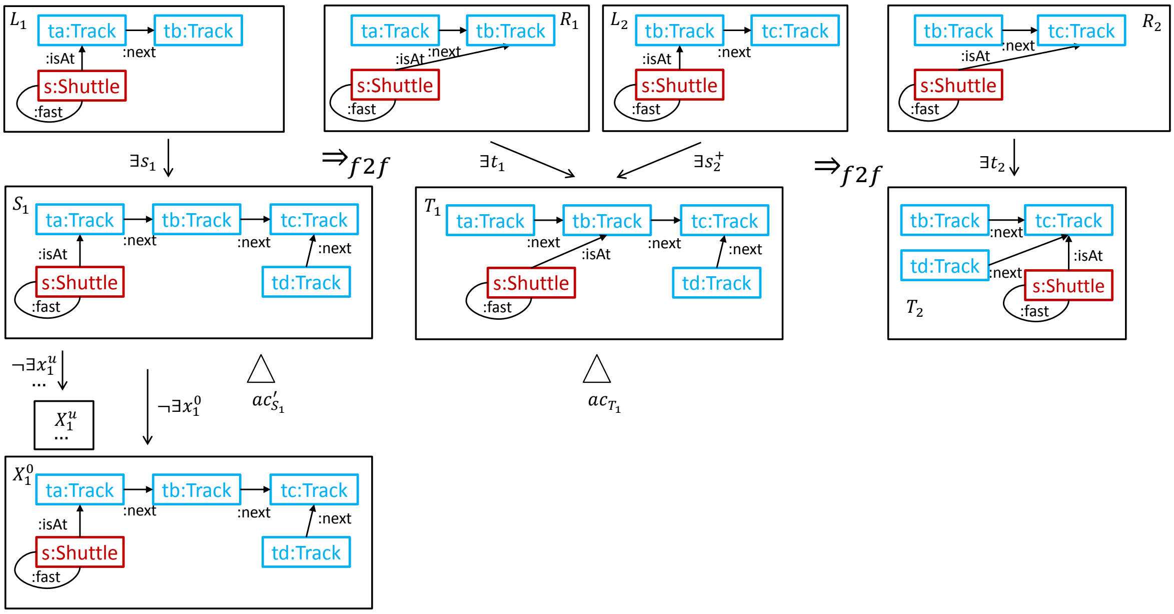


Figure B.16. – Steps SC$_{\text{k}}$-4/5: $seq_2 = \exists(s_1, ac_{S_1}) \Rightarrow_{\text{f2f}} (\exists(t_1, ac_{T_1}), \exists(s_2^+, ac_{T_1})) \Rightarrow_{\text{f2f}} \exists t_2$ with $seq_2 \in \text{Seq}_2^r(\mathcal{R}, F_1)$

Appendix C.

Models and Example Systems

This chapter contains all type graphs, graph rules, and graph constraints used in the running example and case studies in this thesis.

Table C.1. – Systems and their elements

System(s)	Section	TG	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{H}$	$\mathcal{S}$
shuttle-unsafe	C.1.1, p. C-324	C.1(a)	C.1	C.2	C.3	C.4
shuttle-safe	C.1.2, p. C-328	C.5(a)	C.5	C.2	C.3	C.4
shuttle-single-fault-unsafe	C.1.3, p. C-330	C.6(a)	C.6	C.2(a)	C.3, C.6(m)	C.4
shuttle-single-fault-safe	C.1.4, p. C-332	C.7(a)	C.7	C.2(a)	C.3, C.7(m)	C.4
shuttle-brake-late-n	C.1.5, p. C-334	C.8(a)	C.8(c)	C.8(b)	C.8(e)–C.8(k)	–
shuttle-brake-late-prio-n	C.1.5, p. C-334	C.8(a)	C.8(c)-C.8(d)	C.8(b)	C.8(e)–C.8(k)	–
shuttle-attributes-n	C.1.6, p. C-336	C.9(a)	C.9-C.12	C.9(b)	C.13-C.15	–
equiv-s-trans	C.2.1, p. C-344	C.16(a)	C.16	C.21-C.24	C.18-C.20	–
equiv-s-sem	C.2.1, p. C-344	C.16(a)	C.17	C.24	C.18-C.23	–
equiv-trans	C.2.2, p. C-352	C.25(a)	C.25	C.30-C.33	C.27-C.29	–
equiv-sem	C.2.2, p. C-352	C.25(a)	C.26	C.33	C.27-C.32	–
refine-trans	C.2.3, p. C-360	C.25(a)	C.34	C.30, C.31, C.32(a), C.32(c), C.33(a), C.33(c), C.33(e)	C.27, C.28(c)-C.28(h), C.29	–
refine-sem	C.2.3, p. C-360	C.25(a)	C.26	C.33(a), C.33(c), C.33(e)	C.27, C.28(c)-C.28(h), C.29-C.31, C.32(a), C.32(c)	–

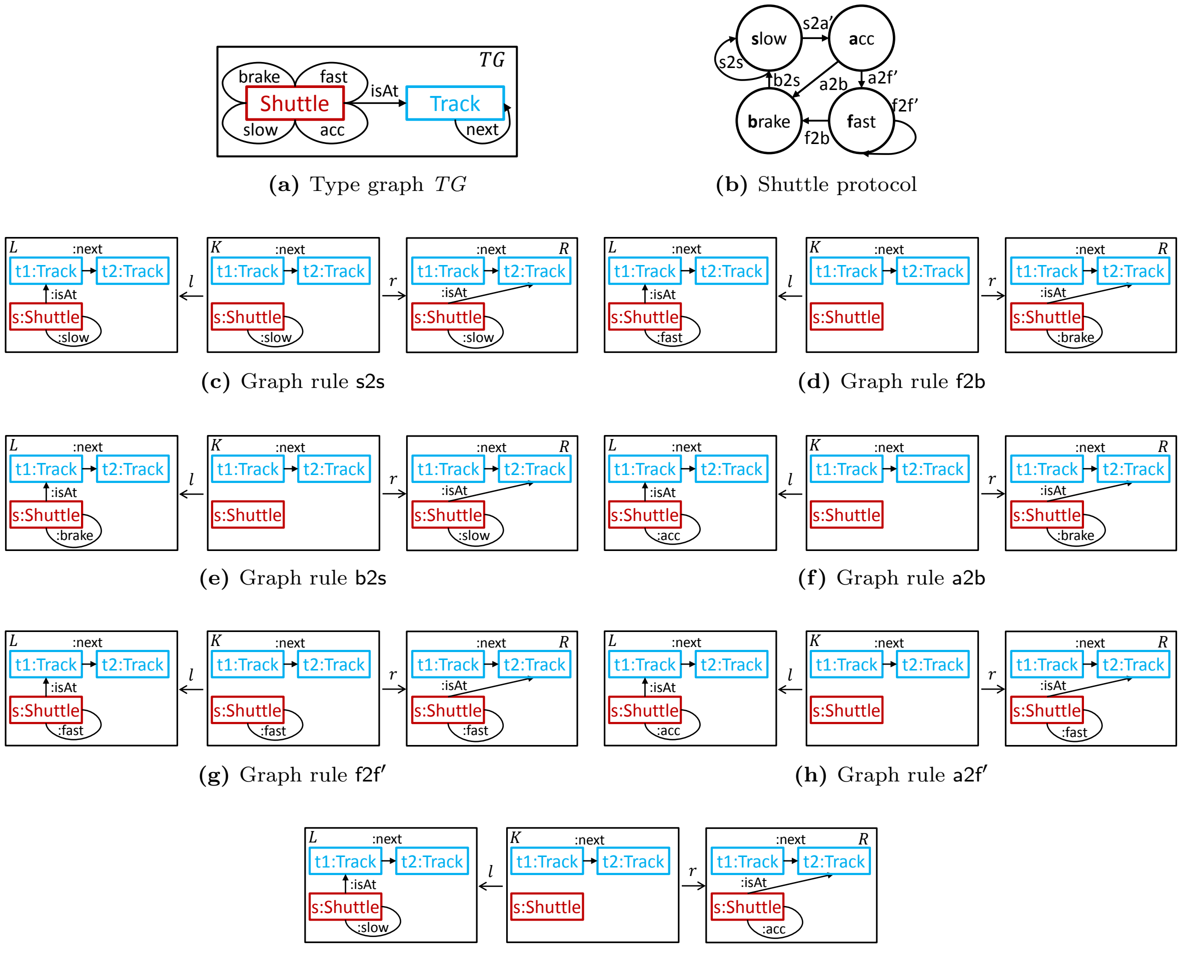
C.1. Shuttle Protocol

C.1.1. Shuttle Protocol shuttle-unsafe

Table C.2. – Type graph, graph rules, and graph constraints of shuttle-unsafe

Element		Fig.	Description
TG		C.1(a)	Connected tracks, shuttles on tracks in different speed modes
$\mathcal{R}$	s2s	C.1(c)	A shuttle moves to a subsequent track in speed mode slow.
	f2b	C.1(d)	A shuttle moves to a subsequent track; mode changes from fast to brake.
	b2s	C.1(e)	A shuttle moves to a subsequent track; mode changes from brake to slow.
	a2b	C.1(f)	A shuttle moves to a subsequent track; mode changes from acc to brake.
	f2f′	C.1(g)	A shuttle moves to a subsequent track in speed mode fast.
	a2f′	C.1(h)	A shuttle moves to a subsequent track; mode changes from acc to fast.
	s2a′	C.1(i)	A shuttle moves to a subsequent track; mode changes from slow to acc.

Element		Fig.	Description
$\mathcal{F} = \bigwedge_{1 \leq i \leq 3} \neg F_i$	$\neg F_1$	C.2(a)	There must not exist a fast shuttle on a switch
	$\neg F_2$	C.2(b)	There must not exist an acc. shuttle on a switch
	$\neg F_3$	C.2(c)	There must not exist a brak(ing) shuttle on a switch
$\mathcal{H} = \bigwedge_{1 \leq j \leq 15} \neg H_j$	$\neg H_1$	C.3(a)	A shuttle cannot be at two tracks at the same time.
	$\neg H_2$	C.3(b)	Two tracks cannot be connected in both directions.
	$\neg H_3$	C.3(c)	Direct predecessors of switches are not connected.
	$\neg H_4$	C.3(d)	Two tracks cannot be connected by parallel next edges.
	$\neg H_5$	C.3(e)	There exists at most one shuttle.
	$\neg H_6$	C.3(f)	A shuttle cannot be in modes fast and slow at once.
	$\neg H_7$	C.3(g)	A shuttle cannot be in modes fast and brake at once.
	$\neg H_8$	C.3(h)	A shuttle cannot be in modes fast and acc at once.
	$\neg H_9$	C.3(i)	A shuttle cannot be in mode fast twice at once.
	$\neg H_{10}$	C.3(j)	A shuttle cannot be in modes brake and slow at once.
	$\neg H_{11}$	C.3(k)	A shuttle cannot be in modes brake and acc at once.
	$\neg H_{12}$	C.3(l)	A shuttle cannot be in mode brake twice at once.
	$\neg H_{13}$	C.3(m)	A shuttle cannot be in modes slow and acc at once.
	$\neg H_{14}$	C.3(n)	A shuttle cannot be in mode slow twice at once.
	$\neg H_{15}$	C.3(o)	A shuttle cannot be in mode acc twice at once.
$\mathcal{S} = \bigwedge_{1 \leq o \leq 3} \neg SC_o$	$\neg SC_1$	C.4(a)	There is no shuttle in mode fast.
	$\neg SC_2$	C.4(b)	There is no shuttle in mode acc.
	$\neg SC_3$	C.4(c)	There is no shuttle on a switch.



(a) Type graph TG

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

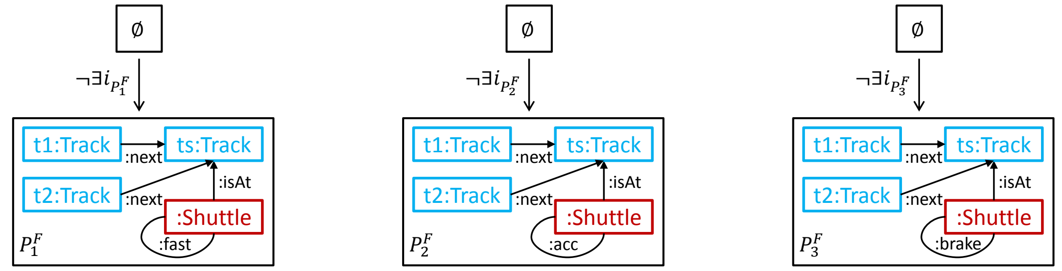
(f) Graph rule a2b

(g) Graph rule f2f′

(h) Graph rule a2f′

(i) Graph rule s2a′

Figure C.1. – Graph transformation system $GTS = (TG, \mathcal{R})$

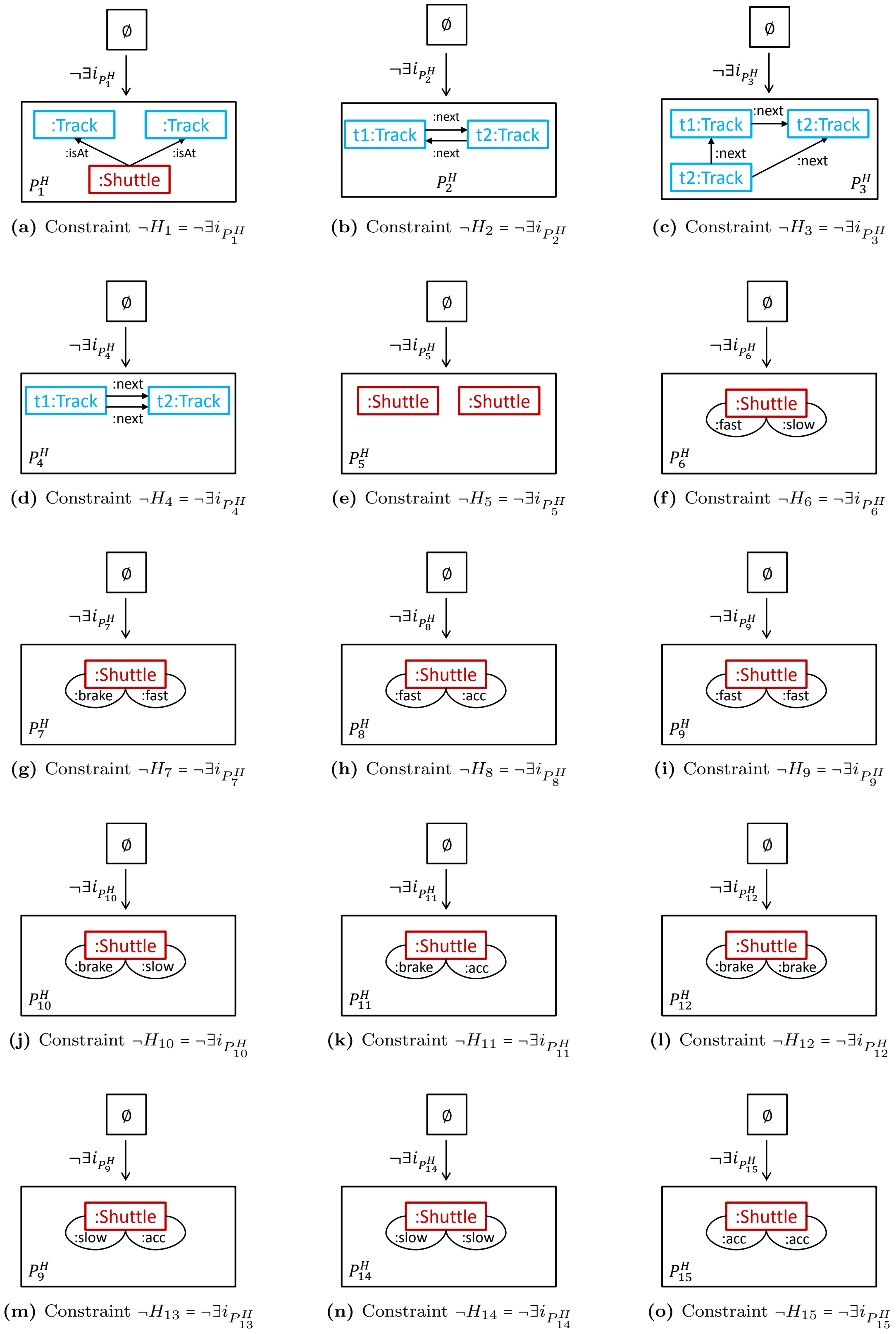


(a) Constraint $\neg F_1 = \neg\exists i_{P_1^F}$

(b) Constraint $\neg F_2 = \neg\exists i_{P_2^F}$

(c) Constraint $\neg F_3 = \neg\exists i_{P_3^F}$

Figure C.2. – Safety property (composed forbidden pattern) $\mathcal{F} = \neg F_1 \wedge \neg F_2 \wedge \neg F_3$



(a) Constraint $\neg H_1 = \neg\exists i_{P_1^H}$

(b) Constraint $\neg H_2 = \neg\exists i_{P_2^H}$

(c) Constraint $\neg H_3 = \neg\exists i_{P_3^H}$

(d) Constraint $\neg H_4 = \neg\exists i_{P_4^H}$

(e) Constraint $\neg H_5 = \neg\exists i_{P_5^H}$

(f) Constraint $\neg H_6 = \neg\exists i_{P_6^H}$

(g) Constraint $\neg H_7 = \neg\exists i_{P_7^H}$

(h) Constraint $\neg H_8 = \neg\exists i_{P_8^H}$

(i) Constraint $\neg H_9 = \neg\exists i_{P_9^H}$

(j) Constraint $\neg H_{10} = \neg\exists i_{P_{10}^H}$

(k) Constraint $\neg H_{11} = \neg\exists i_{P_{11}^H}$

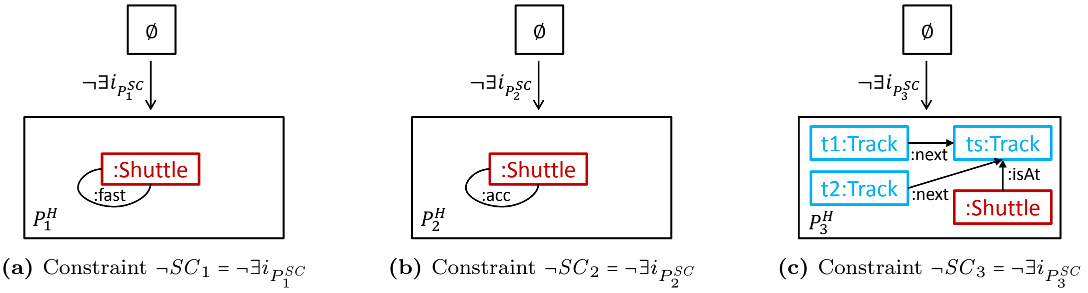
(l) Constraint $\neg H_{12} = \neg\exists i_{P_{12}^H}$

(m) Constraint $\neg H_{13} = \neg\exists i_{P_{13}^H}$

(n) Constraint $\neg H_{14} = \neg\exists i_{P_{14}^H}$

(o) Constraint $\neg H_{15} = \neg\exists i_{P_{15}^H}$

Figure C.3. – Guaranteed constraint (composed guaranteed pattern) $\mathcal{H} = \bigwedge_{1 \le j \le 15} \neg H_j$



(a) Constraint $\neg SC_1 = \neg\exists i_{P_1^{SC}}$ **(b)** Constraint $\neg SC_2 = \neg\exists i_{P_2^{SC}}$ **(c)** Constraint $\neg SC_3 = \neg\exists i_{P_3^{SC}}$

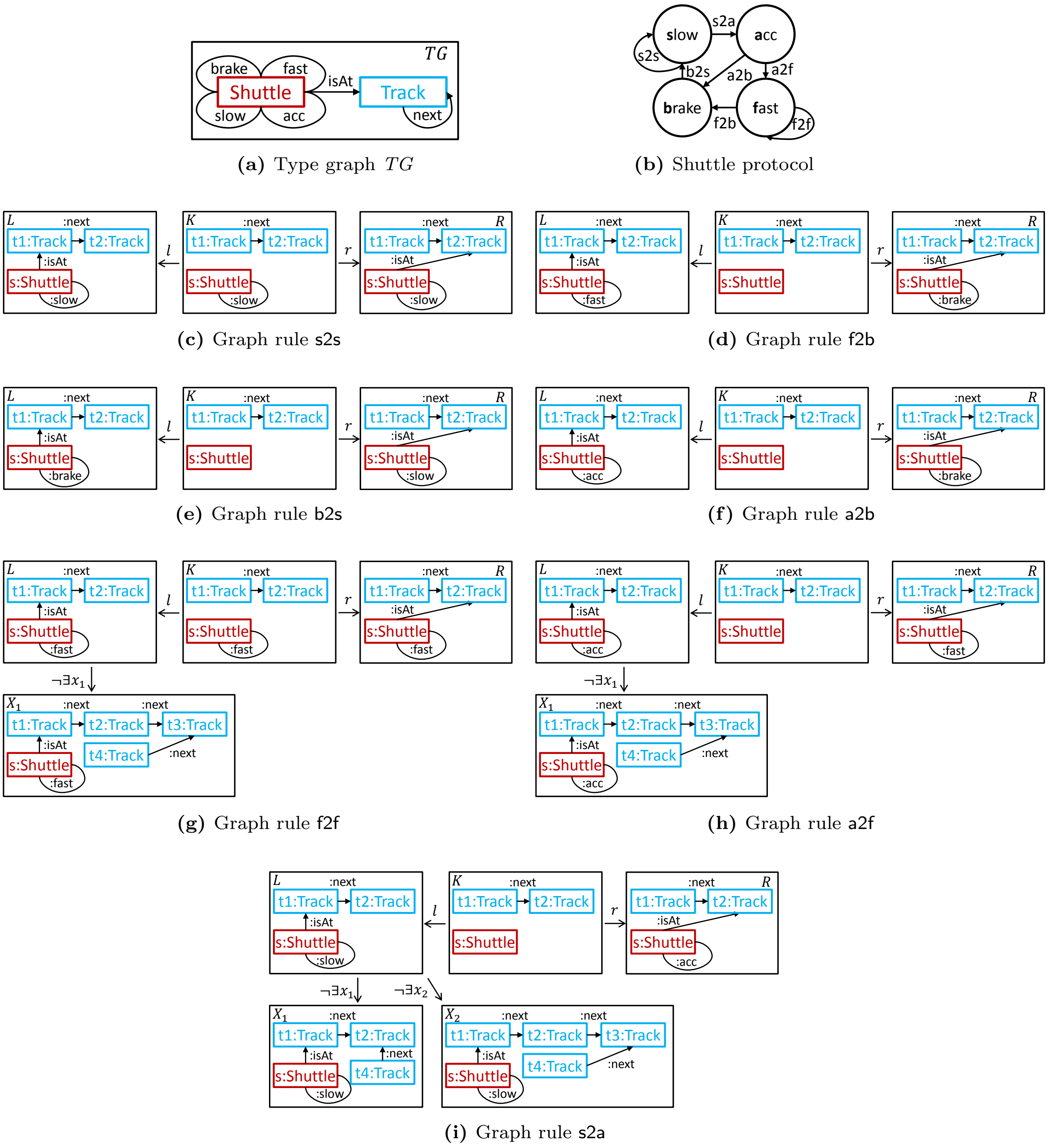
Figure C.4. – Start configuration constraint (composed start configuration pattern) $\mathcal{S} = \bigwedge_{1\leq o\leq 3} \neg SC_o$ (or $\mathcal{S} = \mathcal{H} \wedge \bigwedge_{1\leq o\leq 3} \neg SC_o$)

C.1.2. Shuttle Protocol shuttle-safe

Table C.3. – Type graph, graph rules, and graph constraints of shuttle-safe

Element		Fig.	Description
TG		C.5(a)	Connected tracks, shuttles on tracks in different speed modes.
$\mathcal{R}$	s2s	C.5(c)	A shuttle moves to a subsequent track in speed mode slow.
	f2b	C.5(d)	A shuttle moves to a subsequent track; mode changes from fast to brake.
	b2s	C.5(e)	A shuttle moves to a subsequent track; mode changes from brake to slow.
	a2b	C.5(f)	A shuttle moves to a subsequent track; mode changes from acc to brake.
	f2f	C.5(g)	A shuttle moves to a subsequent track in speed mode fast – unless there is a switch two tracks ahead.
	a2f	C.5(h)	A shuttle moves to a subsequent track; mode changes from acc to fast – unless there is a switch two tracks ahead.
	s2a	C.5(i)	A shuttle moves to a subsequent track; mode changes from slow to acc – unless there is a switch one or two tracks ahead.

Element		Fig.	Description
$\mathcal{F} = \bigwedge_{1 \leq i \leq 3} \neg F_i$	$\neg F_1$	C.2(a)	There must not exist a fast shuttle on a switch
	$\neg F_2$	C.2(b)	There must not exist an acc. shuttle on a switch
	$\neg F_3$	C.2(c)	There must not exist a brak(ing) shuttle on a switch
$\mathcal{H} = \bigwedge_{1 \leq j \leq 15} \neg H_j$	$\neg H_1$	C.3(a)	A shuttle cannot be at two tracks at the same time.
	$\neg H_2$	C.3(b)	Two tracks cannot be connected in both directions.
	$\neg H_3$	C.3(c)	Direct predecessors of switches are not connected.
	$\neg H_4$	C.3(d)	Two tracks cannot be connected by parallel next edges.
	$\neg H_5$	C.3(e)	There exists at most one shuttle.
	$\neg H_6$	C.3(f)	A shuttle cannot be in modes fast and slow at once.
	$\neg H_7$	C.3(g)	A shuttle cannot be in modes fast and brake at once.
	$\neg H_8$	C.3(h)	A shuttle cannot be in modes fast and acc at once.
	$\neg H_9$	C.3(i)	A shuttle cannot be in mode fast twice at once.
	$\neg H_{10}$	C.3(j)	A shuttle cannot be in modes brake and slow at once.
	$\neg H_{11}$	C.3(k)	A shuttle cannot be in modes brake and acc at once.
	$\neg H_{12}$	C.3(l)	A shuttle cannot be in mode brake twice at once.
	$\neg H_{13}$	C.3(m)	A shuttle cannot be in modes slow and acc at once.
	$\neg H_{14}$	C.3(n)	A shuttle cannot be in mode slow twice at once.
	$\neg H_{15}$	C.3(o)	A shuttle cannot be in mode acc twice at once.
$\mathcal{S} = \bigwedge_{1 \leq o \leq 3} \neg SC_o$	$\neg SC_1$	C.4(a)	There is no shuttle in mode fast.
	$\neg SC_2$	C.4(b)	There is no shuttle in mode acc.
	$\neg SC_3$	C.4(c)	There is no shuttle on a switch.



(a) Type graph *TG*

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

(f) Graph rule a2b

(g) Graph rule f2f

(h) Graph rule a2f

(i) Graph rule s2a

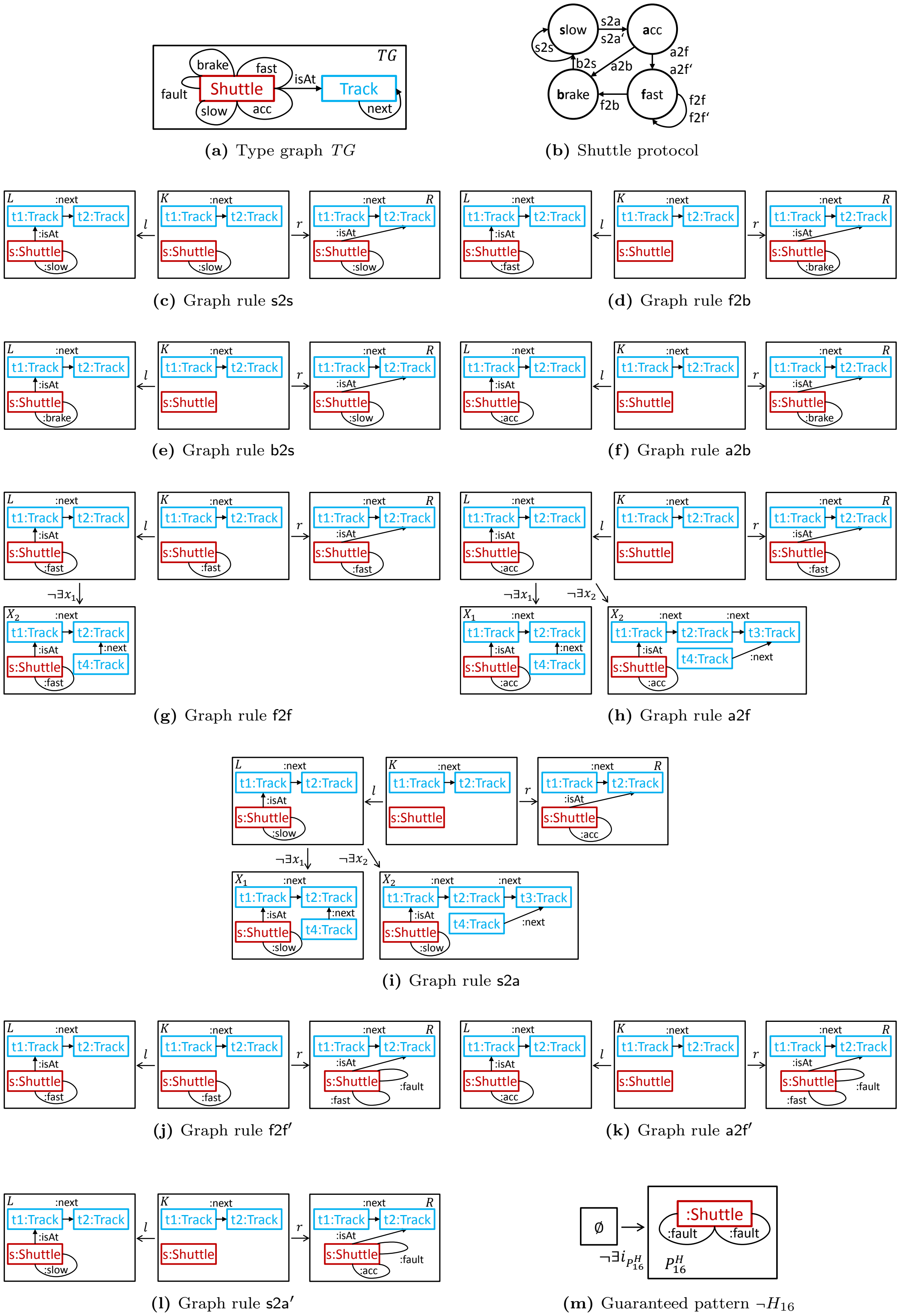
Figure C.5. – Graph transformation system $GTS = (TG, \mathcal{R})$

C.1.3. Shuttle Protocol shuttle-single-fault-unsafe

Table C.4. – Type graph, graph rules, and graph constraints of shuttle-single-fault-unsafe

Element		Fig.	Description
TG		C.6(a)	Connected tracks, shuttles on tracks in different speed modes.
$\mathcal{R}$	s2s	C.6(c)	A shuttle moves to a subsequent track in speed mode slow.
	f2b	C.6(d)	A shuttle moves to a subsequent track; mode changes from fast to brake.
	b2s	C.6(e)	A shuttle moves to a subsequent track; mode changes from brake to slow.
	a2b	C.6(f)	A shuttle moves to a subsequent track; mode changes from acc to brake.
	f2f	C.6(g)	A shuttle moves to a subsequent track in speed mode fast – unless there is a switch one track ahead.
	a2f	C.6(h)	A shuttle moves to a subsequent track; mode changes from acc to fast – unless there is a switch one or two tracks ahead.
	s2a	C.6(i)	A shuttle moves to a subsequent track; mode changes from slow to acc – unless there is a switch one or two tracks ahead.
	f2f	C.6(j)	A shuttle moves to a subsequent track in speed mode fast; sensor fault.
	a2f	C.6(k)	A shuttle moves to a subsequent track; speed mode changes; sensor fault.
	s2a	C.6(l)	A shuttle moves to a subsequent track; speed mode changes; sensor fault.

Element		Fig.	Description
$\mathcal{F} = \neg F_1$	$\neg F_1$	C.2(a)	There must not exist a fast shuttle on a switch.
$\mathcal{H} = \bigwedge_{1 \leq j \leq 16} \neg H_j$	$\neg H_1$	C.3(a)	A shuttle cannot be at two tracks at the same time.
	$\neg H_2$	C.3(b)	Two tracks cannot be connected in both directions.
	$\neg H_3$	C.3(c)	Direct predecessors of switches are not connected.
	$\neg H_4$	C.3(d)	Two tracks cannot be connected by parallel next edges.
	$\neg H_5$	C.3(e)	There exists at most one shuttle.
	$\neg H_6$	C.3(f)	A shuttle cannot be in modes fast and slow at once.
	$\neg H_7$	C.3(g)	A shuttle cannot be in modes fast and brake at once.
	$\neg H_8$	C.3(h)	A shuttle cannot be in modes fast and acc at once.
	$\neg H_9$	C.3(i)	A shuttle cannot be in mode fast twice at once.
	$\neg H_{10}$	C.3(j)	A shuttle cannot be in modes brake and slow at once.
	$\neg H_{11}$	C.3(k)	A shuttle cannot be in modes brake and acc at once.
	$\neg H_{12}$	C.3(l)	A shuttle cannot be in mode brake twice at once.
	$\neg H_{13}$	C.3(m)	A shuttle cannot be in modes slow and acc at once.
	$\neg H_{14}$	C.3(n)	A shuttle cannot be in mode slow twice at once.
	$\neg H_{15}$	C.3(o)	A shuttle cannot be in mode acc twice at once.
	$\neg H_{16}$	C.6(m)	There cannot be more than one sensor fault.
$\mathcal{S} = \bigwedge_{1 \leq o \leq 3} \neg SC_o$	$\neg SC_1$	C.4(a)	There is no shuttle in mode fast.
	$\neg SC_2$	C.4(b)	There is no shuttle in mode acc.
	$\neg SC_3$	C.4(c)	There is no shuttle on a switch



(a) Type graph TG

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

(f) Graph rule a2b

(g) Graph rule f2f

(h) Graph rule a2f

(i) Graph rule s2a

(j) Graph rule f2f′

(k) Graph rule a2f′

(l) Graph rule s2a′

(m) Guaranteed pattern $\neg H_{16}$

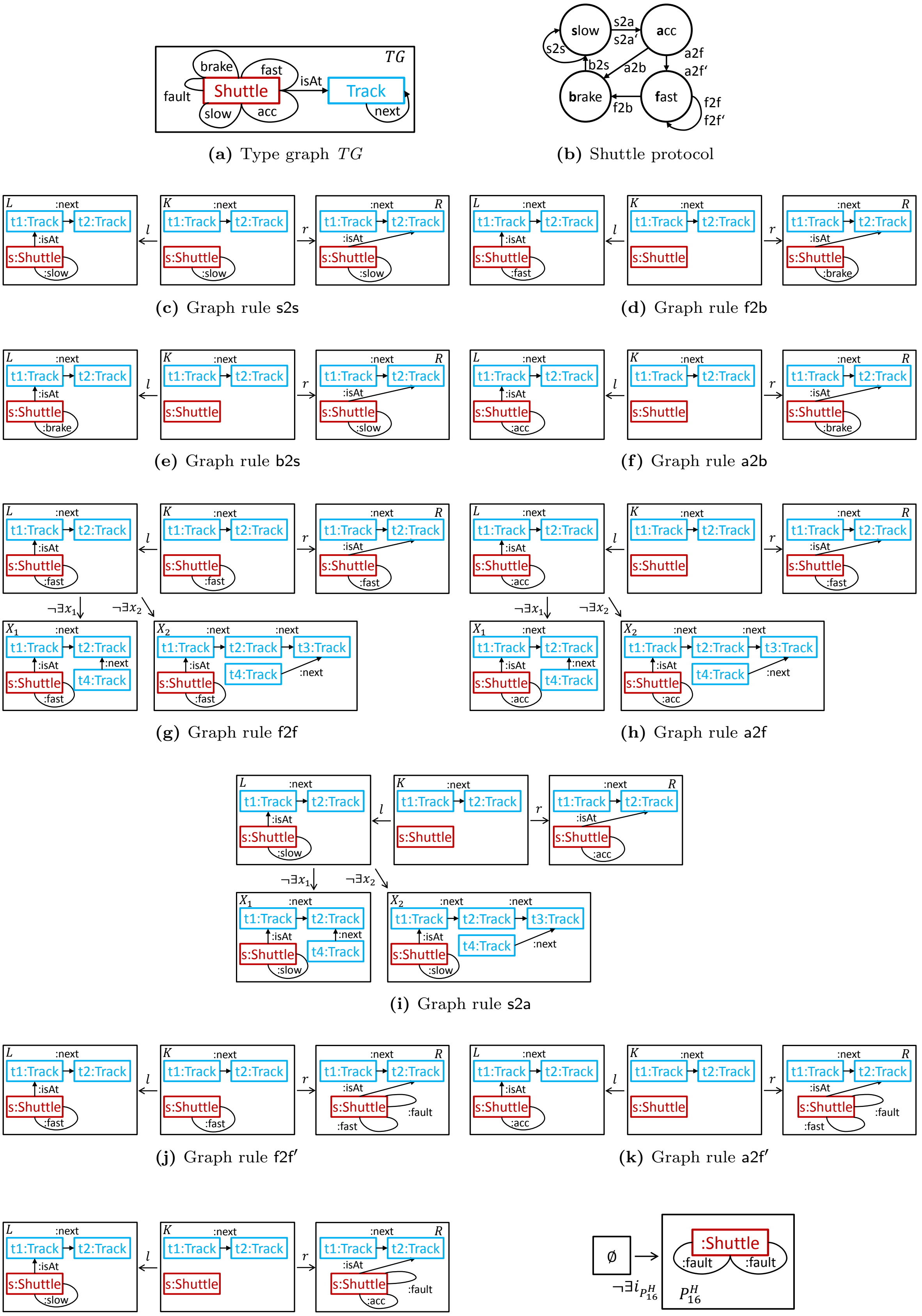
Figure C.6. – Graph transformation system $GTS = (TG, \mathcal{R})$ and guaranteed pattern $\neg H_{16}$

C.1.4. Shuttle Protocol shuttle-single-fault-safe

Table C.5. – Type graph, graph rules, and graph constraints of shuttle-single-fault-unsafe

Element		Fig.	Description
TG		C.7(a)	Connected tracks, shuttles on tracks in different speed modes
$\mathcal{R}$	s2s	C.7(c)	A shuttle moves to a subsequent track in speed mode slow
	f2b	C.7(d)	A shuttle moves to a subsequent track; mode changes from fast to brake
	b2s	C.7(e)	A shuttle moves to a subsequent track; mode changes from brake to slow
	a2b	C.7(f)	A shuttle moves to a subsequent track; mode changes from acc to brake
	f2f	C.7(g)	A shuttle moves to a subsequent track in speed mode fast – unless there is a switch one track ahead
	a2f	C.7(h)	A shuttle moves to a subsequent track; mode changes from acc to fast – unless there is a switch one or two tracks ahead
	s2a	C.7(i)	A shuttle moves to a subsequent track; mode changes from slow to acc – unless there is a switch one or two tracks ahead
	f2f	C.7(j)	A shuttle moves to a subsequent track in speed mode fast; sensor fault
	a2f	C.7(k)	A shuttle moves to a subsequent track; speed mode changes; sensor fault
	s2a	C.7(l)	A shuttle moves to a subsequent track; speed mode changes; sensor fault

Element		Fig.	Description
$\mathcal{F} = \neg F_1$	$\neg F_1$	C.2(a)	There must not exist a fast shuttle on a switch
$\mathcal{H} = \bigwedge_{1 \leq j \leq 16} \neg H_j$	$\neg H_1$	C.3(a)	A shuttle cannot be at two tracks at the same time
	$\neg H_2$	C.3(b)	Two tracks cannot be connected in both directions
	$\neg H_3$	C.3(c)	Direct predecessors of switches are not connected
	$\neg H_4$	C.3(d)	Two tracks cannot be connected by parallel next edges
	$\neg H_5$	C.3(e)	There exists at most one shuttle
	$\neg H_6$	C.3(f)	A shuttle cannot be in modes fast and slow at once
	$\neg H_7$	C.3(g)	A shuttle cannot be in modes fast and brake at once
	$\neg H_8$	C.3(h)	A shuttle cannot be in modes fast and acc at once
	$\neg H_9$	C.3(i)	A shuttle cannot be in mode fast twice at once
	$\neg H_{10}$	C.3(j)	A shuttle cannot be in modes brake and slow at once
	$\neg H_{11}$	C.3(k)	A shuttle cannot be in modes brake and acc at once
	$\neg H_{12}$	C.3(l)	A shuttle cannot be in mode brake twice at once
	$\neg H_{13}$	C.3(m)	A shuttle cannot be in modes slow and acc at once
	$\neg H_{14}$	C.3(n)	A shuttle cannot be in mode slow twice at once
	$\neg H_{15}$	C.3(o)	A shuttle cannot be in mode acc twice at once
	$\neg H_{16}$	C.7(m)	There cannot be more than one sensor fault
$\mathcal{S} = \bigwedge_{1 \leq o \leq 3} \neg SC_o$	$\neg SC_1$	C.4(a)	There is no shuttle in mode fast
	$\neg SC_2$	C.4(b)	There is no shuttle in mode acc
	$\neg SC_3$	C.4(c)	There is no shuttle on a switch



(a) Type graph TG

(b) Shuttle protocol

(c) Graph rule s2s

(d) Graph rule f2b

(e) Graph rule b2s

(f) Graph rule a2b

(g) Graph rule f2f

(h) Graph rule a2f

(i) Graph rule s2a

(j) Graph rule f2f′

(k) Graph rule a2f′

(l) Graph rule s2a′

(m) Guaranteed pattern $\neg H_{16}$

Figure C.7. – Graph transformation system $GTS = (TG, \mathcal{R})$ and guaranteed pattern $\neg H_{16}$

C.1.5. Shuttle Protocol Fragments shuttle-brake-late-n and shuttle-brake-late-prio-n

Table C.6. – Type graph, rules, and constraints of shuttle-brake-late-n fragment

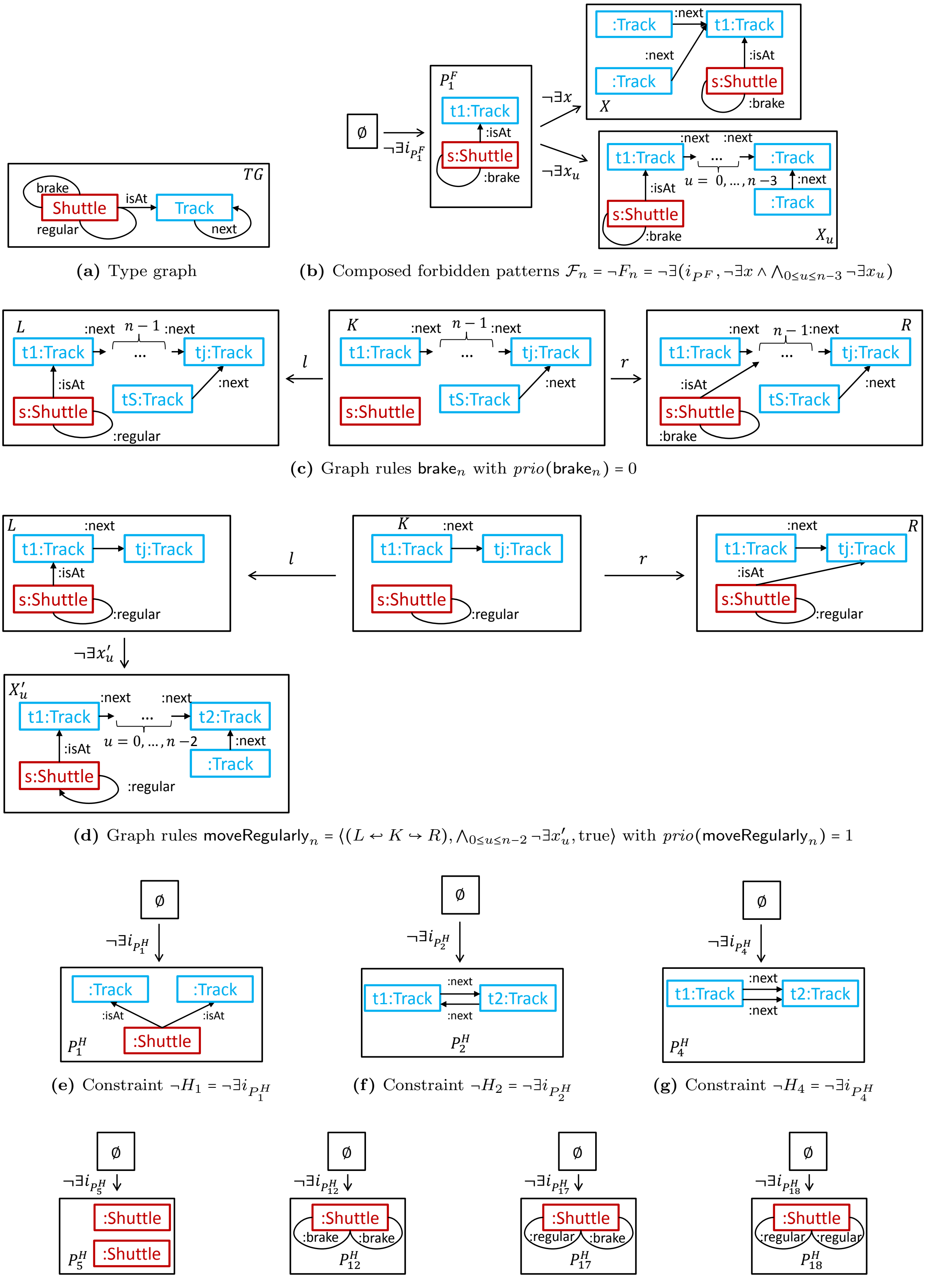
Element		Fig.	Description
TG		C.8(a)	Connected tracks, shuttles on tracks in modes regular or brake.
$\mathcal{R}$	brake$_n$	C.8(c)	A shuttle moves to a subsequent track with a switch n tracks ahead; mode changes from regular to brake; priority 0.

Element		Fig.	Description
$\mathcal{F} = \neg F_n$	$\neg F_n$	C.8(b)	A shuttle must not brake unless there is a switch at most $n-2$ tracks ahead.
$\mathcal{H} = \bigwedge_{j \in J} \neg H_j$	$\neg H_1$	C.8(e)	A shuttle cannot be at two tracks at the same time.
	$\neg H_2$	C.8(f)	Two tracks cannot be connected in both directions.
	$\neg H_4$	C.8(g)	Two tracks cannot be connected by parallel next edges.
	$\neg H_5$	C.8(h)	There exists at most one shuttle.
	$\neg H_{12}$	C.8(i)	A shuttle cannot be in mode brake twice at once.
	$\neg H_{17}$	C.8(j)	A shuttle cannot be in modes brake and regular at once.
	$\neg H_{18}$	C.8(j)	A shuttle cannot be in mode regular twice at once.

Table C.7. – Type graph, rules, and constraints of shuttle-brake-late-prio-n fragment

Element		Fig.	Description
TG		C.8(a)	Connected tracks, shuttles on tracks in modes regular or brake.
$\mathcal{R}$	brake$_n$	C.8(c)	A shuttle moves to a subsequent track with a switch n tracks ahead; mode changes from regular to brake; priority 0.
	moveRegularly$_n$	C.8(d)	A shuttle moves to a subsequent track in speed mode regular – unless a switch is at most $n-1$ track ahead; priority 1.

Element		Fig.	Description
$\mathcal{F} = \neg F_n$	$\neg F_n$	C.8(b)	A shuttle must not brake unless there is a switch at most $n-2$ tracks ahead.
$\mathcal{H} = \bigwedge_{j \in J} \neg H_j$	$\neg H_1$	C.8(e)	A shuttle cannot be at two tracks at the same time.
	$\neg H_2$	C.8(f)	Two tracks cannot be connected in both directions.
	$\neg H_4$	C.8(g)	Two tracks cannot be connected by parallel next edges..
	$\neg H_5$	C.8(h)	There exists at most one shuttle.
	$\neg H_{12}$	C.8(i)	A shuttle cannot be in mode brake twice at once.
	$\neg H_{17}$	C.8(j)	A shuttle cannot be in modes brake and regular at once.
	$\neg H_{18}$	C.8(k)	A shuttle cannot be in mode regular twice at once.



(a) Type graph

(b) Composed forbidden patterns $\mathcal{F}_n = \neg F_n = \neg\exists(i_{P^F}, \neg\exists x \wedge \bigwedge_{0\leq u\leq n-3} \neg\exists x_u)$

(c) Graph rules brake_n with $prio(\mathsf{brake}_n) = 0$

(d) Graph rules $\mathsf{moveRegularly}_n = \langle (L \hookleftarrow K \hookrightarrow R), \bigwedge_{0\leq u\leq n-2} \neg\exists x'_u, \mathrm{true}\rangle$ with $prio(\mathsf{moveRegularly}_n) = 1$

(e) Constraint $\neg H_1 = \neg\exists i_{P_1^H}$

(f) Constraint $\neg H_2 = \neg\exists i_{P_2^H}$

(g) Constraint $\neg H_4 = \neg\exists i_{P_4^H}$

(h) Constraint $\neg H_5$

(i) Constraint $\neg H_{12}$

(j) Constraint $\neg H_{17}$

(k) Constraint $\neg H_{18}$

Figure C.8. – Graph rules brake_n and $\mathsf{moveRegularly}_n$, forbidden patterns F_n, and guaranteed patterns H_j

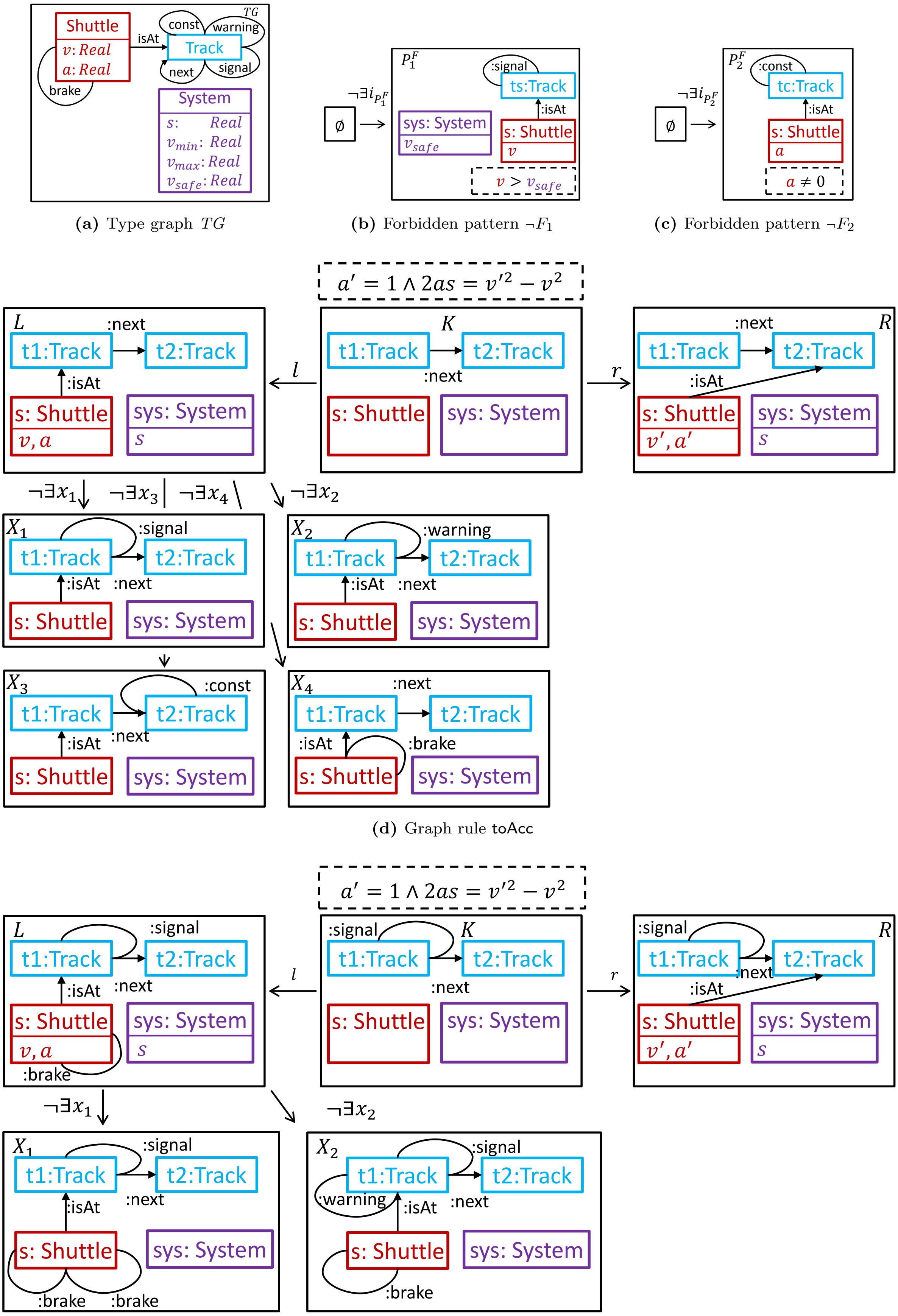
C.1.6. Shuttle Protocol with Attributes (shuttle-attributes$_n$)

Table C.8. – Type graph and rules of shuttle-attributes$_n$

Element		Fig.	Description
TG		C.9(a)	There are connected tracks of a fixed length, some of which are signals, warnings (for signals ahead), or require constant velocity. Shuttles have a velocity (v), acceleration (a), and a braking mode. A system singleton stores maximum (v_{max}) and minimum velocity (v_{min}), maximum velocity to pass a signal safely (v_{safe}), and track length (s).
$\mathcal{R}$	toAcc	C.9(d)	A shuttle moves to a subsequent track and accelerates – unless it is in mode brake, its current track is a signal or warning, or the subsequent track requires constant velocity.
	toAcc-remBrake	C.9(e)	A shuttle moves from a track with a signal to a subsequent track, accelerates, and leaves braking mode – unless another brake flag is set or its current track is a warning.
	toDec	C.10(a)	A shuttle moves to a subsequent track and decelerates – unless its current track is a signal or warning or the subsequent track requires constant velocity.
	toDec-warning	C.10(b)	A shuttle moves to a subsequent track, decelerates, and enters braking mode – unless its current track is a signal or the subsequent track requires constant velocity.
	toDec-remBrake	C.10(c)	A shuttle moves from a track with a signal to a subsequent track, decelerates, and leaves braking mode – unless its current track is a warning.
	toSteady	C.11(a)	A shuttle moves to a subsequent track and holds its velocity – unless its current track is a signal or warning or it is in braking mode.
	toSteady-const-warning	C.11(b)	A shuttle moves to a subsequent track that requires constant velocity, holds its velocity, and enters braking mode – unless its current track is a signal.
	toSteady-const	C.12(a)	A shuttle moves to a subsequent track that requires constant velocity and holds its velocity – unless its current track is a warning.
	toSteady-remBrake	C.12(b)	A shuttle moves from a track with a signal to a subsequent track, holds its velocity, and leaves braking mode – unless its current track is a warning or a another brake flag is set.

Table C.9. – Patterns of shuttle-attributes$_n$

Element		Fig.	Description
$\mathcal{F} = \neg F_1$	$\neg F_1$	C.9(b)	A shuttle must not pass a signal with a velocity greater that the designated safe velocity (v_{safe}).
	$\neg F_2$	C.9(c)	A shuttle must not pass a track that requires constant velocity (const) with non-constant velocity.
$\mathcal{H} = \bigwedge_{j \in J} \neg H_j$	$\neg H_1$	C.13(a)	A shuttle cannot be at two tracks at the same time.
	$\neg H_2$	C.13(b)	Two tracks cannot be connected in both directions.
	$\neg H_4$	C.13(c)	Two tracks cannot be connected by parallel next edges.
	$\neg H_5$	C.13(d)	There exists at most one shuttle.
	$\neg H_{19}$	C.13(e)	There exists at most one (global) system node.
	$\neg H_{20}$	C.14(a)	Shuttles have fixed acceleration values when accelerating. (1), decelerating (−2), or holding their velocity (0)
	$\neg H_{21}$	C.14(b)	Shuttles cannot exceed a fixed maximum velocity (v_{max}). or fall below a fixed minimum velocity (v_{min})
	$\neg H_{22}$	C.14(c)	Tracks have a fixed length (s) of 500, minimum (v_{min}) and maximum velocity (v_{max}) of shuttles is 2 and 50, respectively, and the maximum velocity to safely pass a signal (v_{safe}) is 20. This pattern's attribute constraint can be modified to change system parameters, if, for instance, track length or requirements/restrictions on shuttle velocities change.
	$\neg H_3$	C.15(a)	Direct predecessors of switches cannot be connected.
	$\neg H_{23}$	C.15(b)	A signal cannot be followed by a track that requires constant velocity.
	$\neg H_{24}$	C.15(c)	A track that requires constant velocity cannot be followed by a track that also requires constant velocity.
	$\neg H_{25}$	C.15(d)	A track cannot have both a signal and a warning.
	$\neg H_{26,u}$	C.15(e)	These are $n-2$ patterns; together, they express that there are are no cycles of length n or less.
	$\neg H_{27,u}$	C.15(f)	These are $n - 1$ patterns; together, they express that two signals cannot be $n - 2$ or fewer tracks apart.
	$\neg H_{28,n}$	C.15(g)	All sequences of tracks of length $n + 1$ that end with a signal have a warning on their first track. In other words, each signal requires a warning on each track that is n tracks ahead of the signal.



(a) Type graph *TG*

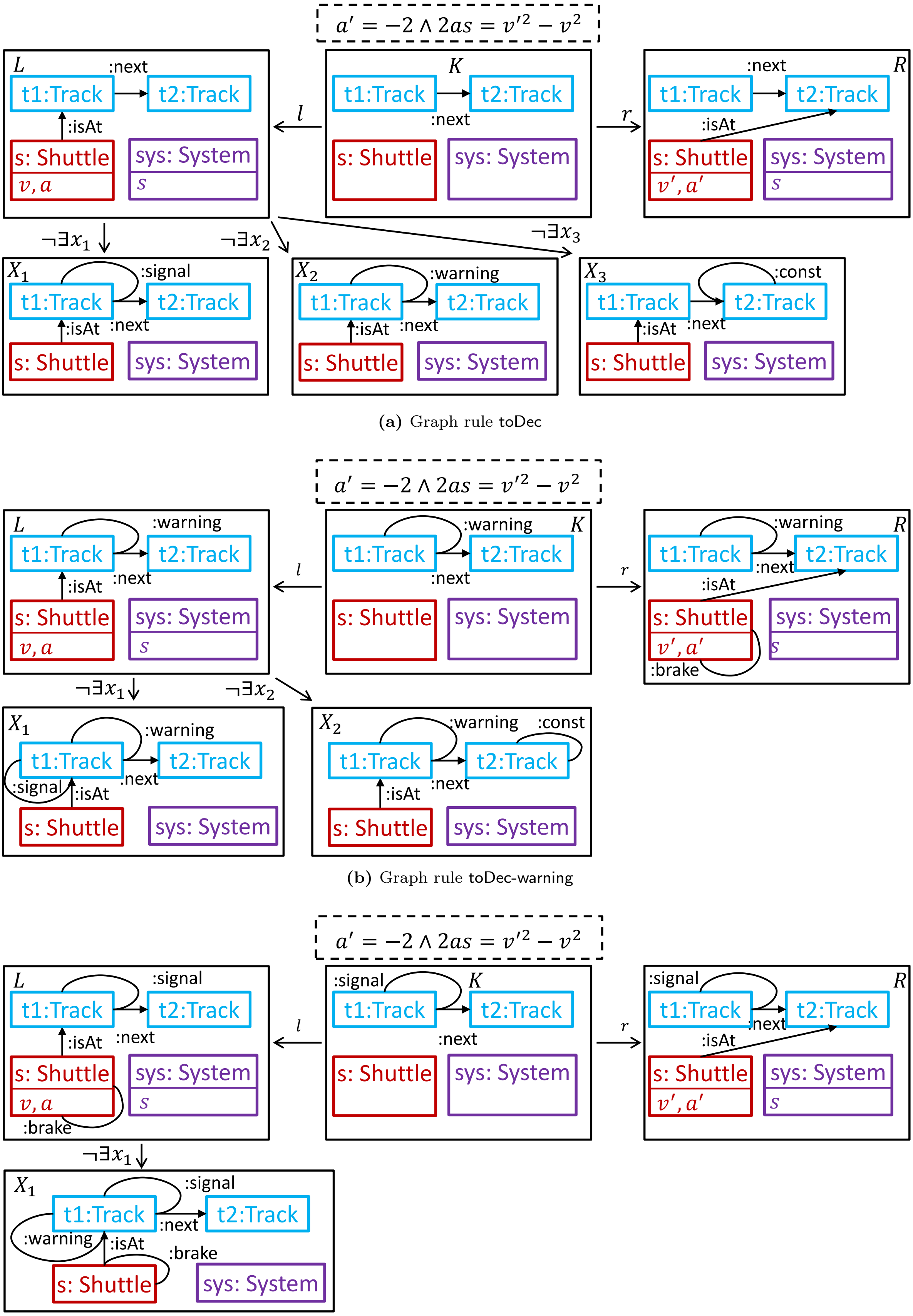
(b) Forbidden pattern $\neg F_1$

(c) Forbidden pattern $\neg F_2$

(d) Graph rule toAcc

(e) Graph rule toAcc-remBrake

Figure C.9. – Type graph, forbidden pattern, and rules toAcc and toAcc-remBrake

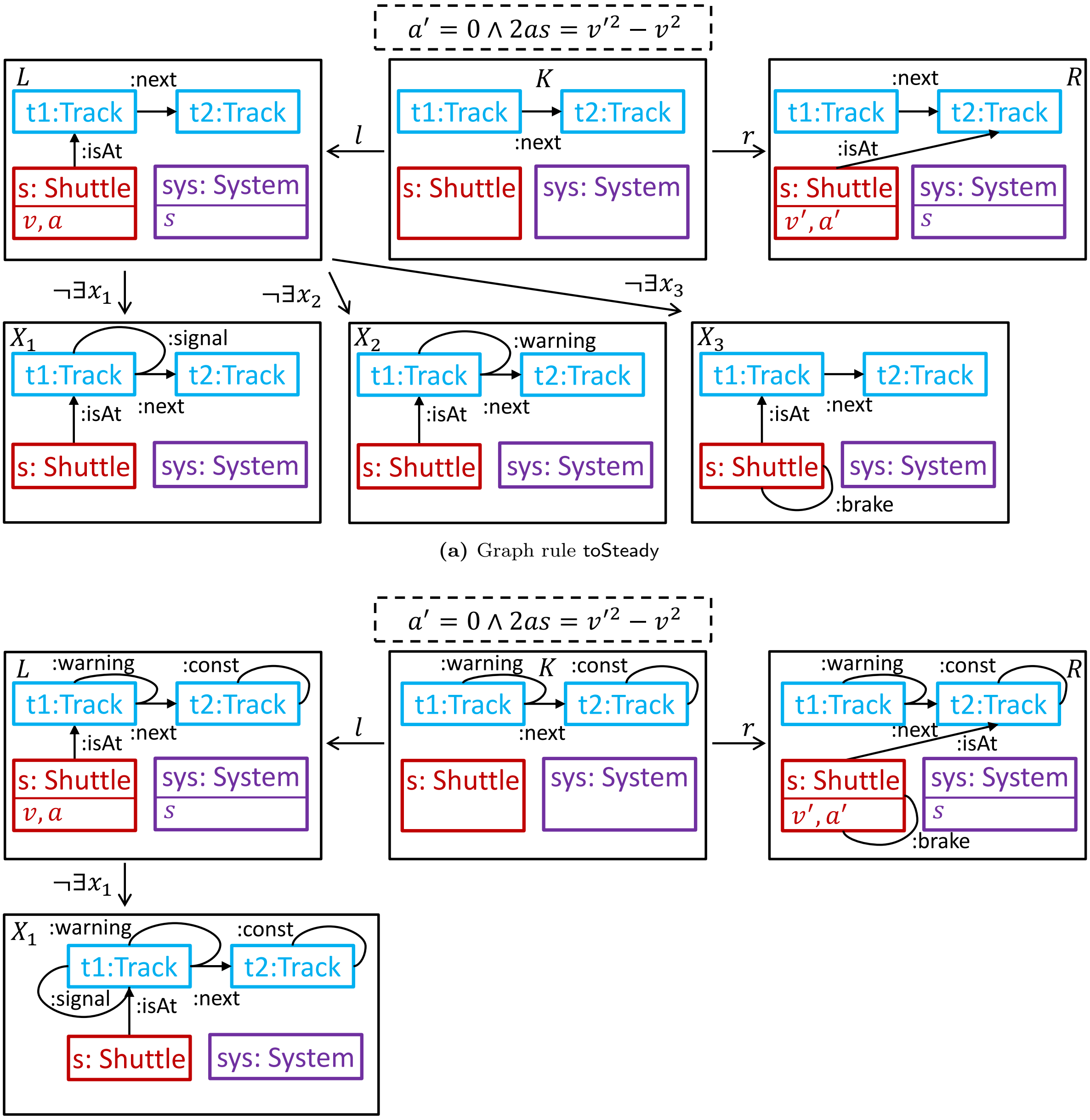


(a) Graph rule toDec

(b) Graph rule toDec-warning

(c) Graph rule toDec-remBrake

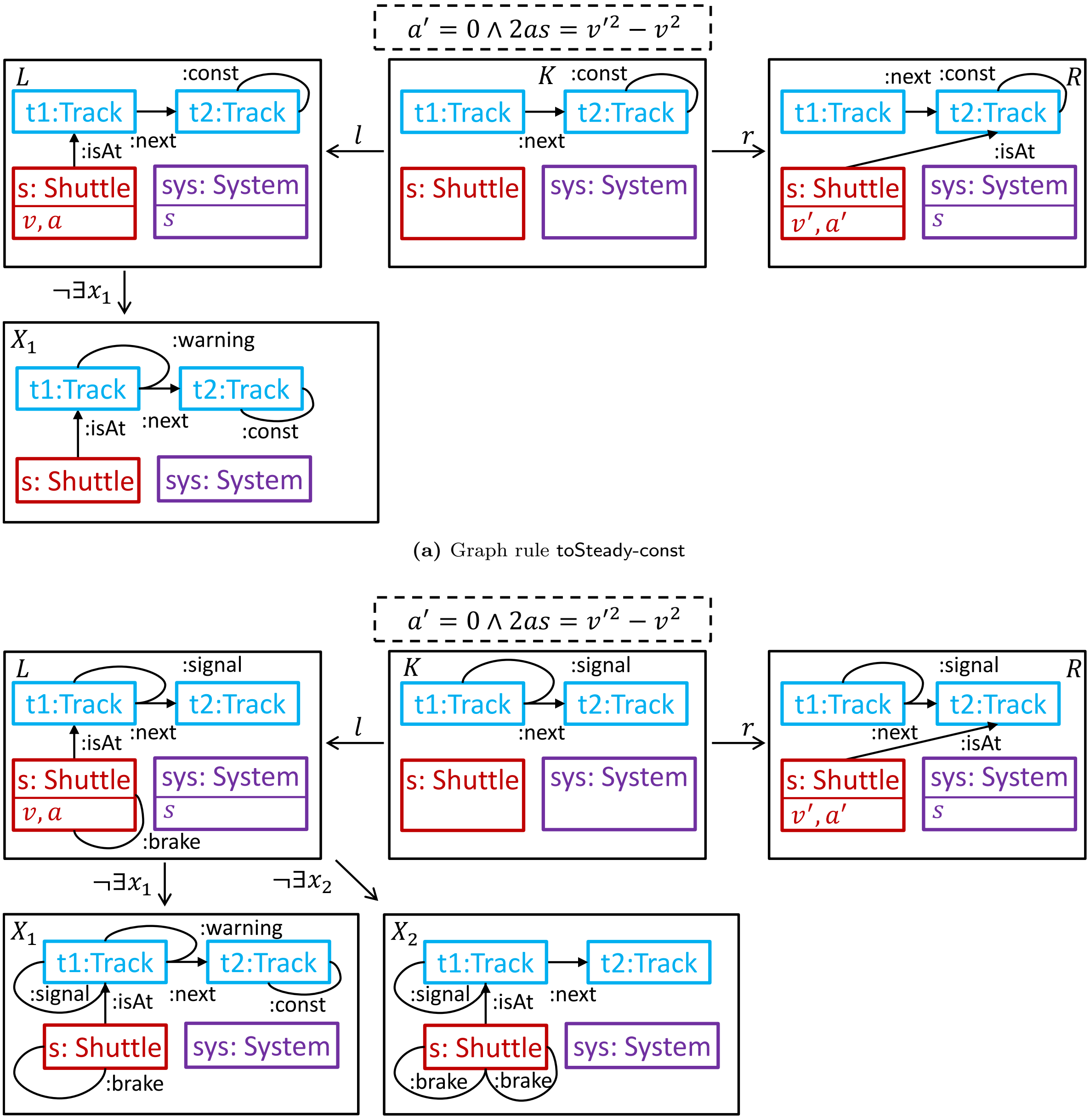
Figure C.10. – Graph rules toDec, toDec-warning, and toDec-remBrake



(a) Graph rule toSteady

(b) Graph rule toSteady-const-warning

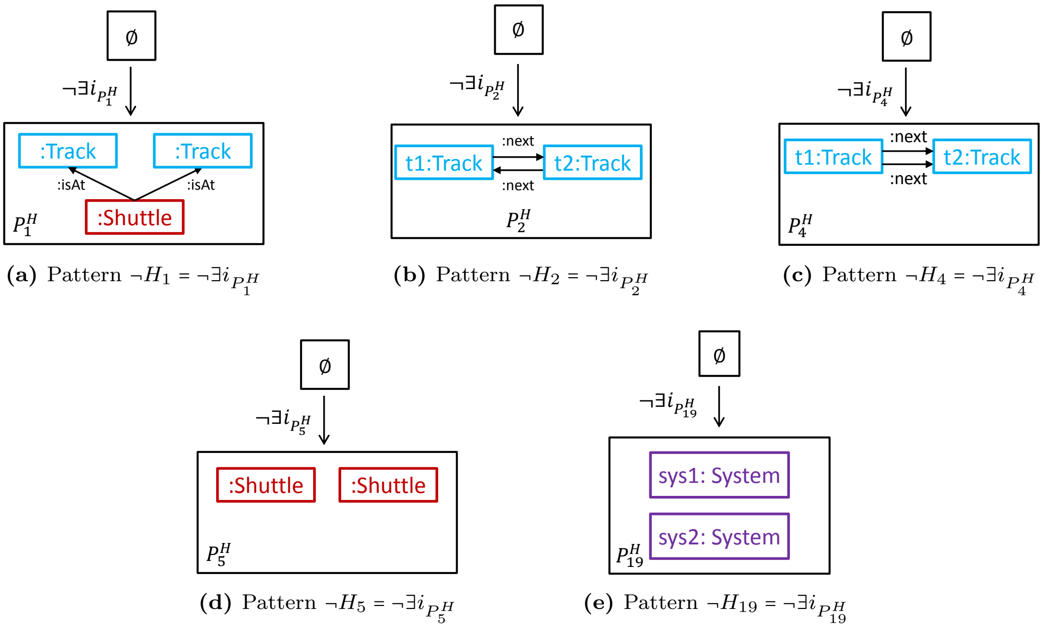
Figure C.11. – Graph rules toSteady and toSteady-const-warning



(a) Graph rule toSteady-const

(b) Graph rule toSteady-remBrake

Figure C.12. – Graph rules toSteady-const and toSteady-remBrake



(a) Pattern $\neg H_1 = \neg\exists i_{P_1^H}$

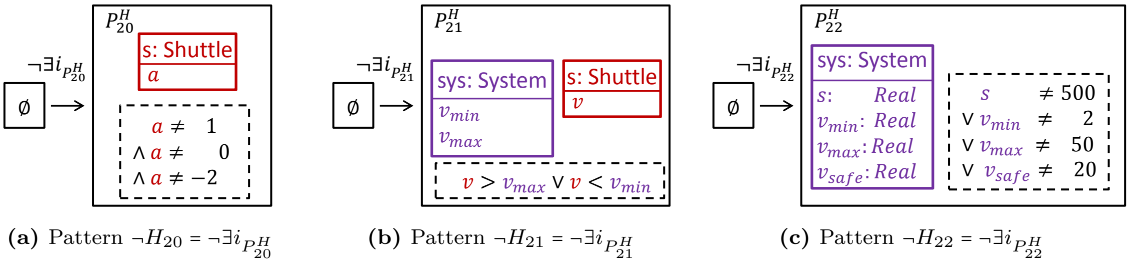
(b) Pattern $\neg H_2 = \neg\exists i_{P_2^H}$

(c) Pattern $\neg H_4 = \neg\exists i_{P_4^H}$

(d) Pattern $\neg H_5 = \neg\exists i_{P_5^H}$

(e) Pattern $\neg H_{19} = \neg\exists i_{P_{19}^H}$

Figure C.13. – Guaranteed patterns addressing cardinality constraints

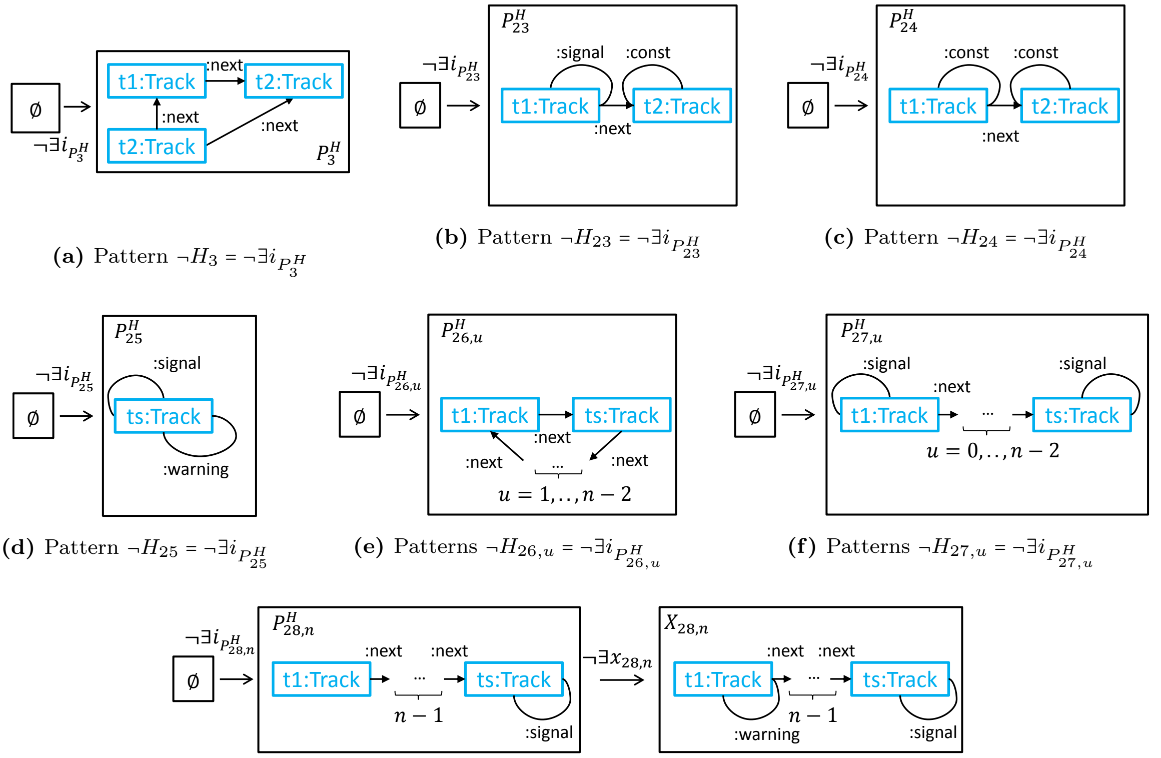


(a) Pattern $\neg H_{20} = \neg\exists i_{P_{20}^H}$

(b) Pattern $\neg H_{21} = \neg\exists i_{P_{21}^H}$

(c) Pattern $\neg H_{22} = \neg\exists i_{P_{22}^H}$

Figure C.14. – Guaranteed patterns addressing attribute values and system parameters



(a) Pattern $\neg H_3 = \neg\exists i_{P^H_3}$

(b) Pattern $\neg H_{23} = \neg\exists i_{P^H_{23}}$

(c) Pattern $\neg H_{24} = \neg\exists i_{P^H_{24}}$

(d) Pattern $\neg H_{25} = \neg\exists i_{P^H_{25}}$

(e) Patterns $\neg H_{26,u} = \neg\exists i_{P^H_{26,u}}$

(f) Patterns $\neg H_{27,u} = \neg\exists i_{P^H_{27,u}}$

(g) Pattern $\neg H_{28,n} = \neg\exists(i_{P^H_{28,n}}, \neg\exists x_{28,n})$

Figure C.15. – Guaranteed patterns addressing track topology

C.2. Behavior Preservation of Model Transformations

C.2.1. Equivalence – equiv-s-trans and equiv-s-sem

Table C.10. – Elements of equiv-s-trans

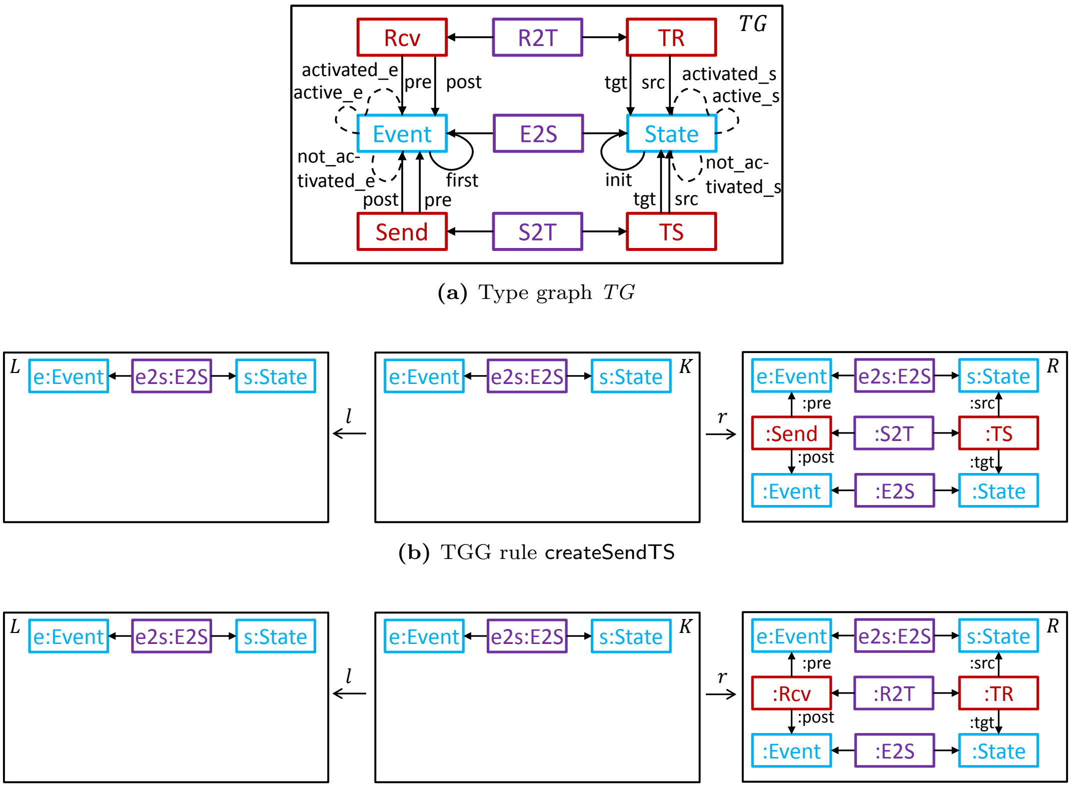
Element		Fig.	Description
TG		C.16(a)	There are events (in a sequence chart), connected by Send and Rcv elements. Similarly, states (in an automaton) are connected by transitions. Correspondence elements connect corresponding elements of the source and target modeling languages.
$\mathcal{R}$	createSendTS	C.16(b)	Creates a message that is sent when an event is triggered and its subsequent event and a corresponding transition with a target state.
	createRcvTR	C.16(c)	Creates a message that is received when an event is triggered and its subsequent event and a corresponding transition with a target state.

Element		Fig.	Description
$\mathcal{F} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}_S^{GTS}$	C.21	Runtime restrictions for semantics of source models.
	$\mathcal{C}_T^{GTS}$	C.22	Runtime restrictions for semantics of target models.
	$\mathcal{C}_{MT}$	C.23	Model transformation constraint, which establishes correspondence properties about the static structure of source and target models.
	$\mathcal{C}_{Pair} \wedge \mathcal{C}_{RT}$	C.24	Pair constraint and runtime constraint required to show the existence of a bimiulation relation.
$\mathcal{H} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}_S$	C.18	Cardinality and type graph constraints for the source modeling language (source component of the type graph).
	$\mathcal{C}_T$	C.19	Cardinality and type graph constraints for the target modeling language (target component of the type graph).
	$\mathcal{C}_{TGG}$	C.20	Cardinality and type graph constraints for correspondence elements (correspondence component of the type graph).

Table C.11. – Elements of equiv-s-sem

Element		Fig.	Description
TG		C.16(a)	There are events (in a sequence chart), connected by Send and Rcv elements. Similarly, states (in an automaton) are connected by transitions. Correspondence elements connect corresponding elements of the source and target modeling languages.
$\mathcal{R}$	initE-initS	C.17(a)	Initializes the sequence chart's first event and the automaton's inital state.
	send-fireTS	C.17(b)	Sends a message, fires a sending transition, and marks the subsequent event and state as active.
	rcv-fireTR	C.17(c)	Receives a message, fires a receiving transition, and marks the subsequent event and state as active.

Element		Fig.	Description
$\mathcal{F} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}_{Pair} \wedge \mathcal{C}_{RT}$	C.24	Pair constraint and runtime constraint required to show the existence of a bimiulation relation.
$\mathcal{H} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}_S$	C.18	Cardinality and type graph constraints for the source modeling language (source component of the type graph).
	$\mathcal{C}_T$	C.19	Cardinality and type graph constraints for the target modeling language (target component of the type graph).
	$\mathcal{C}_{TGG}$	C.20	Cardinality and type graph constraints for correspondence elements (correspondence component of the type graph).
	$\mathcal{C}_S^{GTS}$	C.21	Runtime restrictions for semantics of source models.
	$\mathcal{C}_T^{GTS}$	C.22	Runtime restrictions for semantics of target models.
	$\mathcal{C}_{MT}$	C.23	Model transformation constraint, which establishes correspondence properties about the static structure of source and target models.

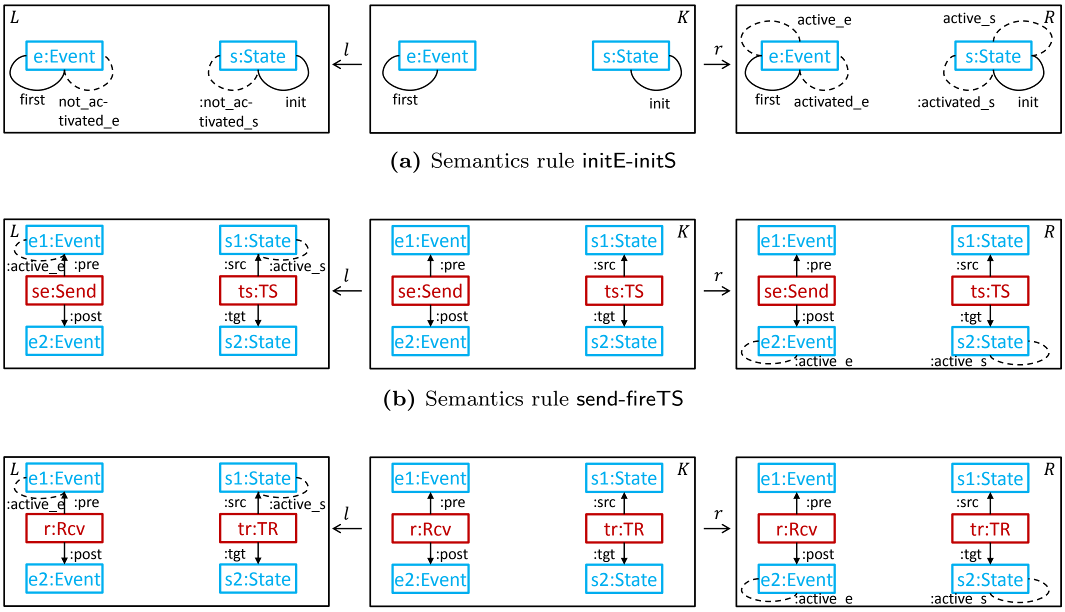


(a) Type graph TG

(b) TGG rule createSendTS

(c) TGG rule createRcvTR

Figure C.16. – Type graph and TGG rules $\mathcal{R}$ = {createSendTS, createRcvTR}

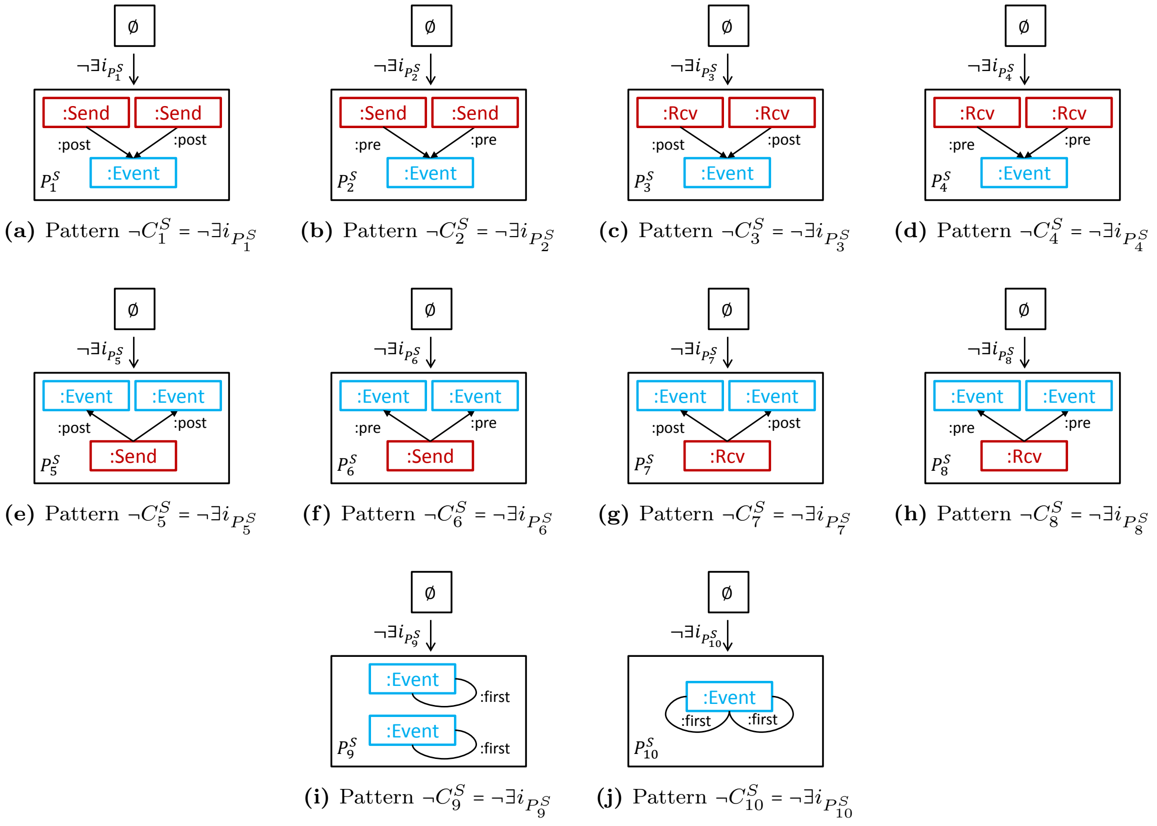


(a) Semantics rule initE-initS

(b) Semantics rule send-fireTS

(c) Semantics rule rcv-fireTR

Figure C.17. – Pair rules $\mathcal{P}(l_s, l_t)$ = {initE-initS, send-fireTS, rcv-fireTR}

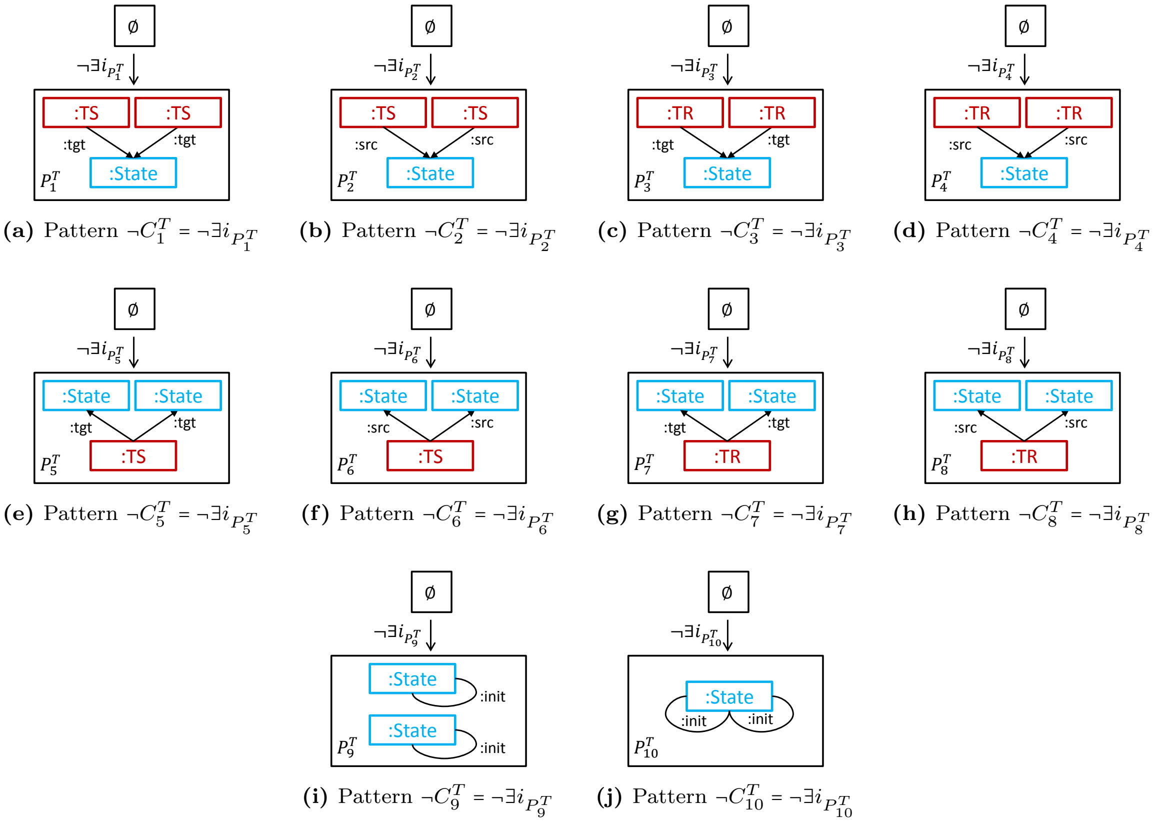

∅
¬∃i
:Send
:Rcv
:Event
:post
:pre
:first
P_1^S P_2^S P_3^S P_4^S P_5^S P_6^S P_7^S P_8^S P_9^S P_{10}^S

(a) Pattern $\neg C_1^S = \neg\exists i_{P_1^S}$ (b) Pattern $\neg C_2^S = \neg\exists i_{P_2^S}$ (c) Pattern $\neg C_3^S = \neg\exists i_{P_3^S}$ (d) Pattern $\neg C_4^S = \neg\exists i_{P_4^S}$

(e) Pattern $\neg C_5^S = \neg\exists i_{P_5^S}$ (f) Pattern $\neg C_6^S = \neg\exists i_{P_6^S}$ (g) Pattern $\neg C_7^S = \neg\exists i_{P_7^S}$ (h) Pattern $\neg C_8^S = \neg\exists i_{P_8^S}$

(i) Pattern $\neg C_9^S = \neg\exists i_{P_9^S}$ (j) Pattern $\neg C_{10}^S = \neg\exists i_{P_{10}^S}$

Figure C.18. – Composed guaranteed pattern $\mathcal{C}_S$

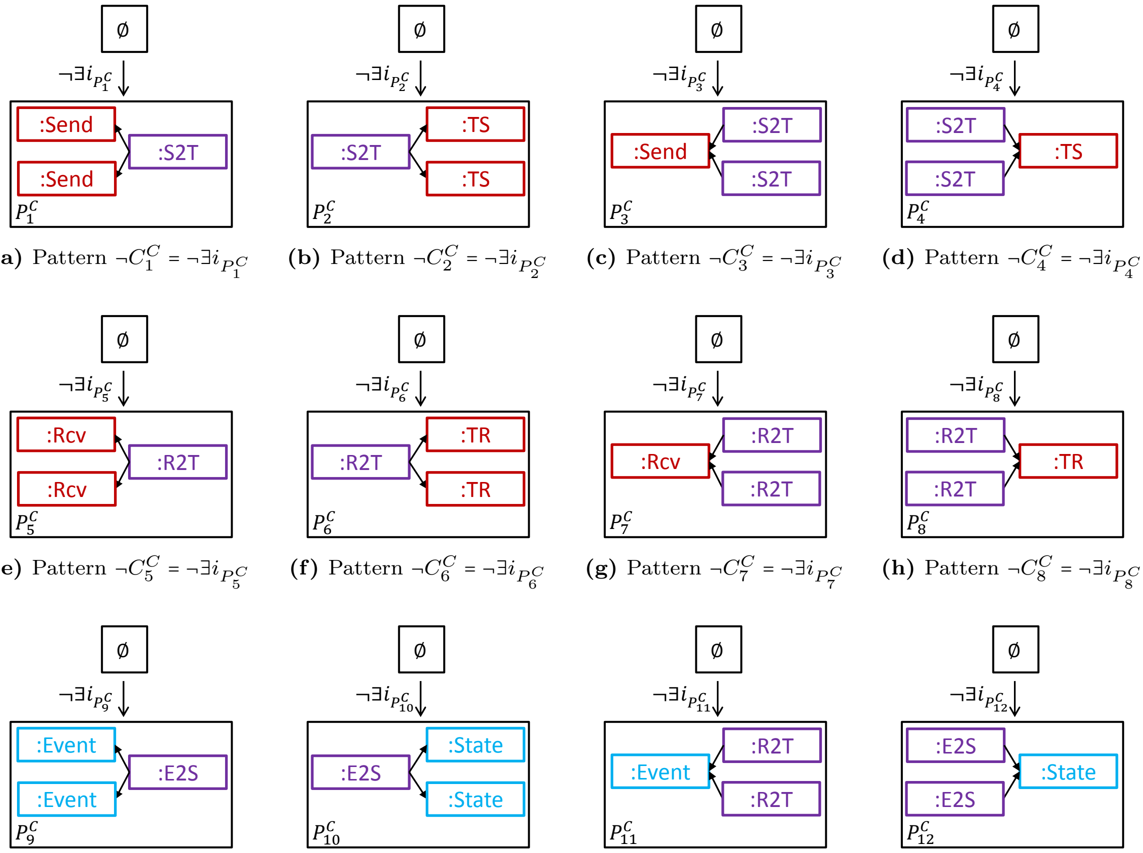

∅
¬∃i
:TS
:TR
:State
:tgt
:src
:init
P_1^T P_2^T P_3^T P_4^T P_5^T P_6^T P_7^T P_8^T P_9^T P_{10}^T

(a) Pattern $\neg C_1^T = \neg\exists i_{P_1^T}$ (b) Pattern $\neg C_2^T = \neg\exists i_{P_2^T}$ (c) Pattern $\neg C_3^T = \neg\exists i_{P_3^T}$ (d) Pattern $\neg C_4^T = \neg\exists i_{P_4^T}$

(e) Pattern $\neg C_5^T = \neg\exists i_{P_5^T}$ (f) Pattern $\neg C_6^T = \neg\exists i_{P_6^T}$ (g) Pattern $\neg C_7^T = \neg\exists i_{P_7^T}$ (h) Pattern $\neg C_8^T = \neg\exists i_{P_8^T}$

(i) Pattern $\neg C_9^T = \neg\exists i_{P_9^T}$ (j) Pattern $\neg C_{10}^T = \neg\exists i_{P_{10}^T}$

Figure C.19. – Composed guaranteed pattern $\mathcal{C}_T$



(a) Pattern $\neg C_1^C = \neg\exists i_{P_1^C}$

(b) Pattern $\neg C_2^C = \neg\exists i_{P_2^C}$

(c) Pattern $\neg C_3^C = \neg\exists i_{P_3^C}$

(d) Pattern $\neg C_4^C = \neg\exists i_{P_4^C}$

(e) Pattern $\neg C_5^C = \neg\exists i_{P_5^C}$

(f) Pattern $\neg C_6^C = \neg\exists i_{P_6^C}$

(g) Pattern $\neg C_7^C = \neg\exists i_{P_7^C}$

(h) Pattern $\neg C_8^C = \neg\exists i_{P_8^C}$

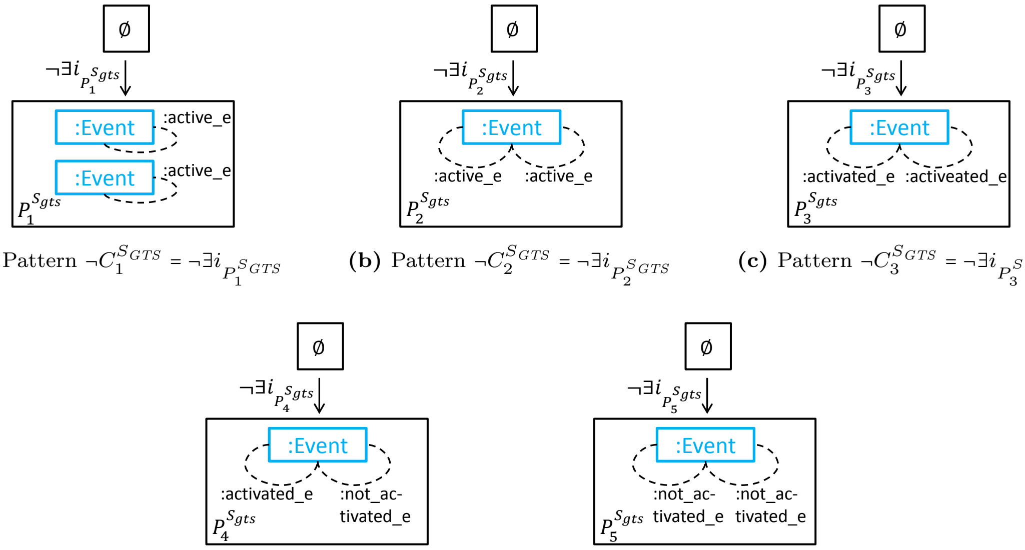
(i) Pattern $\neg C_9^C = \neg\exists i_{P_9^C}$

(j) Pattern $\neg C_{10}^C = \neg\exists i_{P_{10}^C}$

(k) Pattern $\neg C_{11}^C = \neg\exists i_{P_{11}^C}$

(l) Pattern $\neg C_{12}^C = \neg\exists i_{P_{12}^C}$

Figure C.20. – Fragments of composed guaranteed pattern $\mathcal{C}_{TGG}$



(a) Pattern $\neg C_1^{S_{GTS}} = \neg\exists i_{P_1^{S_{GTS}}}$

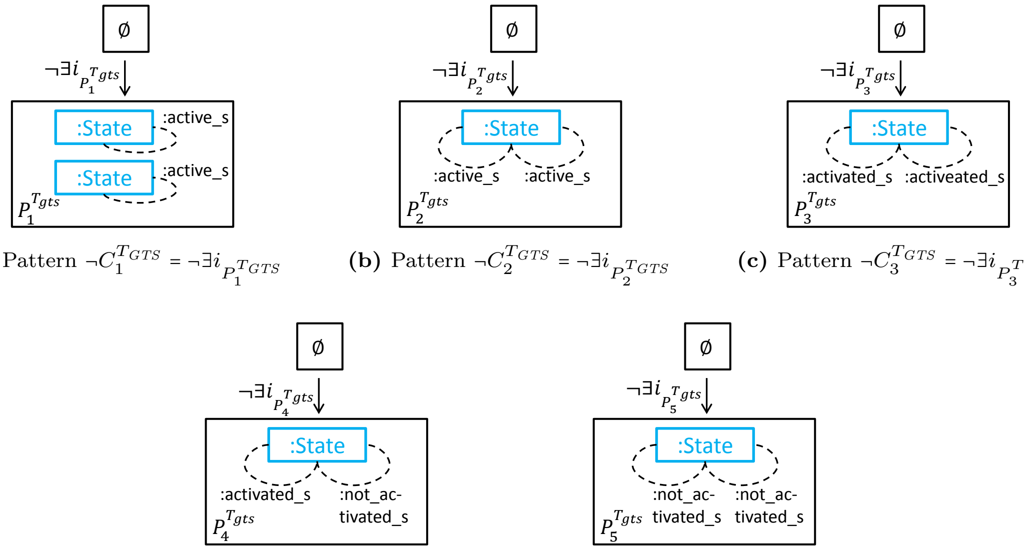
(b) Pattern $\neg C_2^{S_{GTS}} = \neg\exists i_{P_2^{S_{GTS}}}$

(c) Pattern $\neg C_3^{S_{GTS}} = \neg\exists i_{P_3^{S_{GTS}}}$

(d) Pattern $\neg C_4^{S_{GTS}} = \neg\exists i_{P_4^{S_{GTS}}}$

(e) Pattern $\neg C_5^{S_{GTS}} = \neg\exists i_{P_5^{S_{GTS}}}$

Figure C.21. – Composed pattern $\mathcal{C}_S^{GTS}$



(a) Pattern $\neg C_1^{T_{GTS}} = \neg\exists i_{P_1^{T_{GTS}}}$

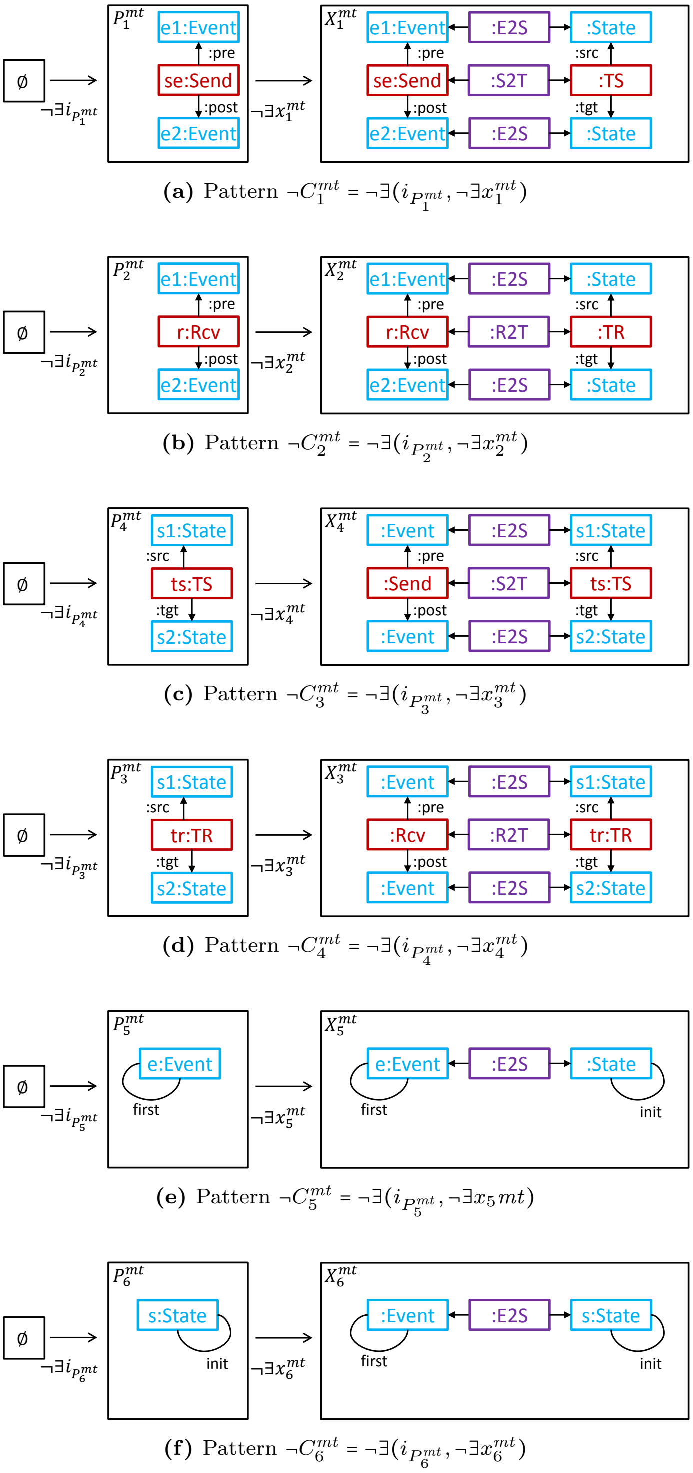
(b) Pattern $\neg C_2^{T_{GTS}} = \neg\exists i_{P_2^{T_{GTS}}}$

(c) Pattern $\neg C_3^{T_{GTS}} = \neg\exists i_{P_3^{T_{GTS}}}$

(d) Pattern $\neg C_4^{T_{GTS}} = \neg\exists i_{P_4^{T_{GTS}}}$

(e) Pattern $\neg C_5^{T_{GTS}} = \neg\exists i_{P_5^{T_{GTS}}}$

Figure C.22. – Composed pattern $\mathcal{C}_T^{GTS}$



(a) Pattern $\neg C_1^{mt} = \neg\exists(i_{P_1^{mt}}, \neg\exists x_1^{mt})$

(b) Pattern $\neg C_2^{mt} = \neg\exists(i_{P_2^{mt}}, \neg\exists x_2^{mt})$

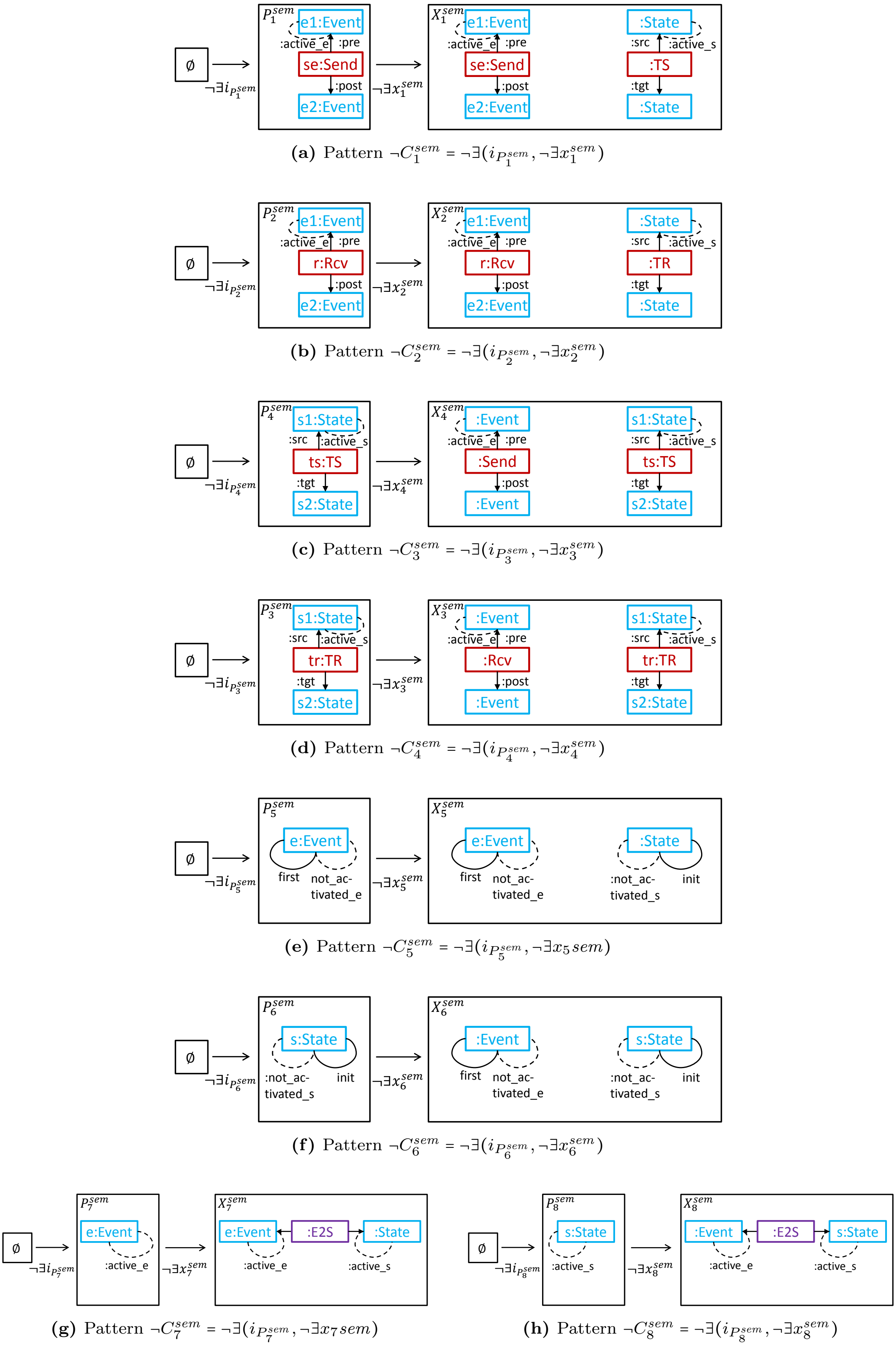
(c) Pattern $\neg C_3^{mt} = \neg\exists(i_{P_3^{mt}}, \neg\exists x_3^{mt})$

(d) Pattern $\neg C_4^{mt} = \neg\exists(i_{P_4^{mt}}, \neg\exists x_4^{mt})$

(e) Pattern $\neg C_5^{mt} = \neg\exists(i_{P_5^{mt}}, \neg\exists x_5 mt)$

(f) Pattern $\neg C_6^{mt} = \neg\exists(i_{P_6^{mt}}, \neg\exists x_6^{mt})$

Figure C.23. – Composed pattern $\mathcal{C}_{MT}$



(a) Pattern $\neg C_1^{sem} = \neg\exists(i_{P_1^{sem}}, \neg\exists x_1^{sem})$

(b) Pattern $\neg C_2^{sem} = \neg\exists(i_{P_2^{sem}}, \neg\exists x_2^{sem})$

(c) Pattern $\neg C_3^{sem} = \neg\exists(i_{P_3^{sem}}, \neg\exists x_3^{sem})$

(d) Pattern $\neg C_4^{sem} = \neg\exists(i_{P_4^{sem}}, \neg\exists x_4^{sem})$

(e) Pattern $\neg C_5^{sem} = \neg\exists(i_{P_5^{sem}}, \neg\exists x_5 sem)$

(f) Pattern $\neg C_6^{sem} = \neg\exists(i_{P_6^{sem}}, \neg\exists x_6^{sem})$

(g) Pattern $\neg C_7^{sem} = \neg\exists(i_{P_7^{sem}}, \neg\exists x_7 sem)$

(h) Pattern $\neg C_8^{sem} = \neg\exists(i_{P_8^{sem}}, \neg\exists x_8^{sem})$

Figure C.24. – Composed pattern $\mathcal{C}_{Pair} \wedge \mathcal{C}_{RT}$

C.2.2. Equivalence – equiv-trans and equiv-sem

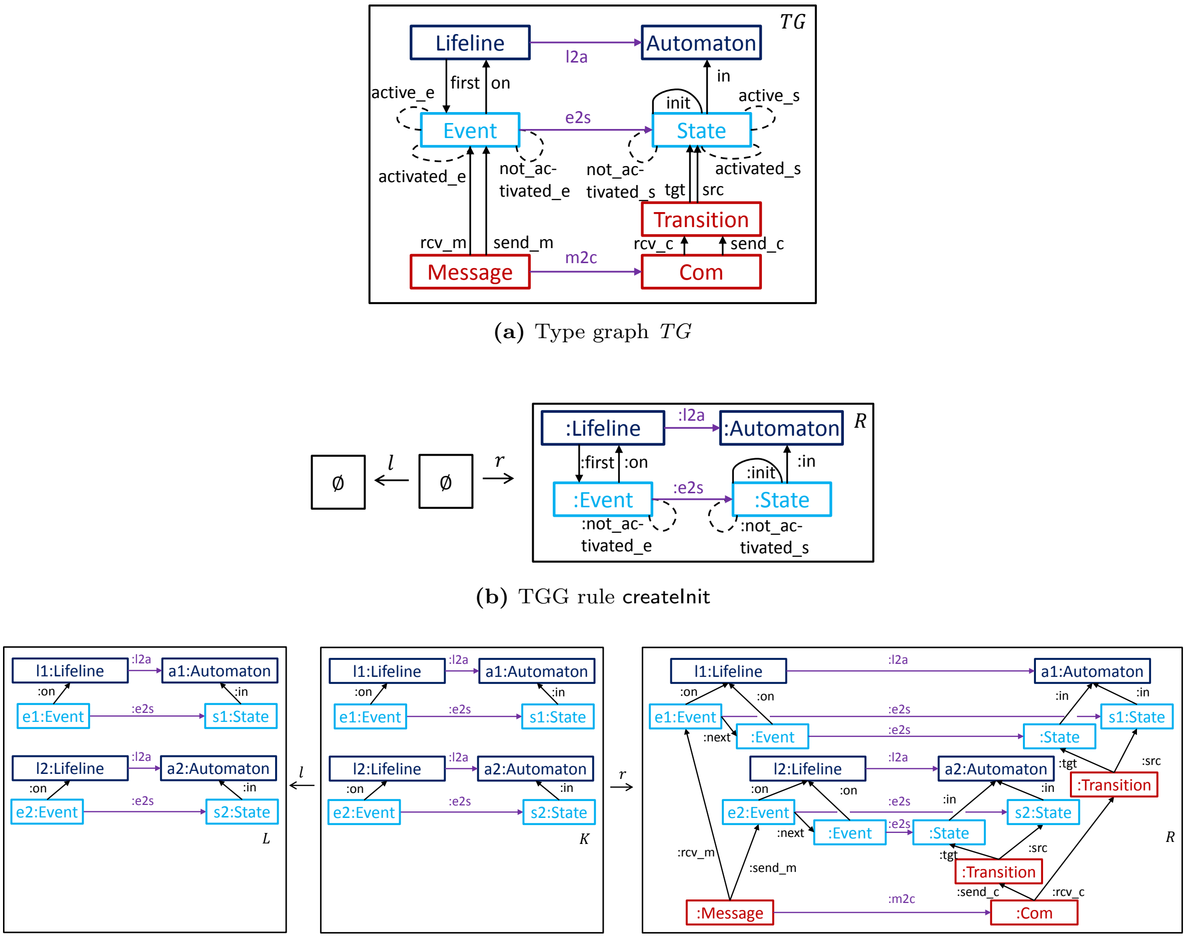
Table C.12. – Elements of equiv-trans

Element		Fig.	Description
TG		C.25(a)	There are events on lifelines, connected by messages that can be received and sent. Similarly, states in a system of communicating automata are connected by communicating transitions. Correspondence elements connect corresponding elements of the source and target modeling languages.
$\mathcal{R}$	createInit	C.25(b)	Creates a lifeline and corresponding automata with an initial event and state, respectively.
	createMsgCom	C.25(c)	Creates a message that can be received and sent by events and creates two corresponding communicating transitions. The rule also creates subsequent events and states.

Element		Fig.	Description
$\mathcal{F} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}^{GTS}_{S}$	C.30	Runtime restrictions for semantics of source models.
	$\mathcal{C}^{GTS}_{T}$	C.31	Runtime restrictions for semantics of target models.
	$\mathcal{C}^{Cor}_{MT}$	C.32	Model transformation constraint, which establishes correspondence properties about the static structure of source and target models.
	$\mathcal{C}^{Cor}_{Pair} \wedge \mathcal{C}_{RT}$	C.33	Pair constraint and runtime constraint required to show the existence of a bimiulation relation.
$\mathcal{H} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}_{S}$	C.27	Cardinality and type graph constraints for the source modeling language (source component of the type graph).
	$\mathcal{C}_{T}$	C.28	Cardinality and type graph constraints for the target modeling language (target component of the type graph).
	$\mathcal{C}_{TGG}$	C.29	Cardinality and type graph constraints for correspondence elements (correspondence component of the type graph).

Table C.13. – Elements of equiv-sem

Element		Fig.	Description
TG		C.25(a)	There are events on lifelines, connected by messages that can be received and sent. Similarly, states in a system of communicating automata are connected by communicating transitions. Correspondence elements connect corresponding elements of the source and target modeling languages.
$\mathcal{R}$	initE-initS	C.26(a)	Initializes a sequence chart's first event and the corresponding automaton's inital state.
	send-fire	C.26(b)	Sends a message and fires the corresponding communicating transitions; also marks subsequent events and states as active.
$\mathcal{F} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}^{Cor}_{Pair} \wedge \mathcal{C}_{RT}$	C.33	Pair constraint and runtime constraint required to show the existence of a bimiulation relation.
$\mathcal{H} = \bigwedge_{\mathcal{C}} \neg C^{\cdots}_{\cdots}$	$\mathcal{C}_S$	C.27	Cardinality and type graph constraints for the source modeling language (source component of the type graph).
	$\mathcal{C}_T$	C.28	Cardinality and type graph constraints for the target modeling language (target component of the type graph).
	$\mathcal{C}_{TGG}$	C.29	Cardinality and type graph constraints for correspondence elements (correspondence component of the type graph).
	$\mathcal{C}^{GTS}_S$	C.30	Runtime restrictions for semantics of source models.
	$\mathcal{C}^{GTS}_T$	C.31	Runtime restrictions for semantics of target models.
	$\mathcal{C}^{Cor}_{MT}$	C.32	Model transformation constraint, which establishes correspondence properties about the static structure of source and target models.

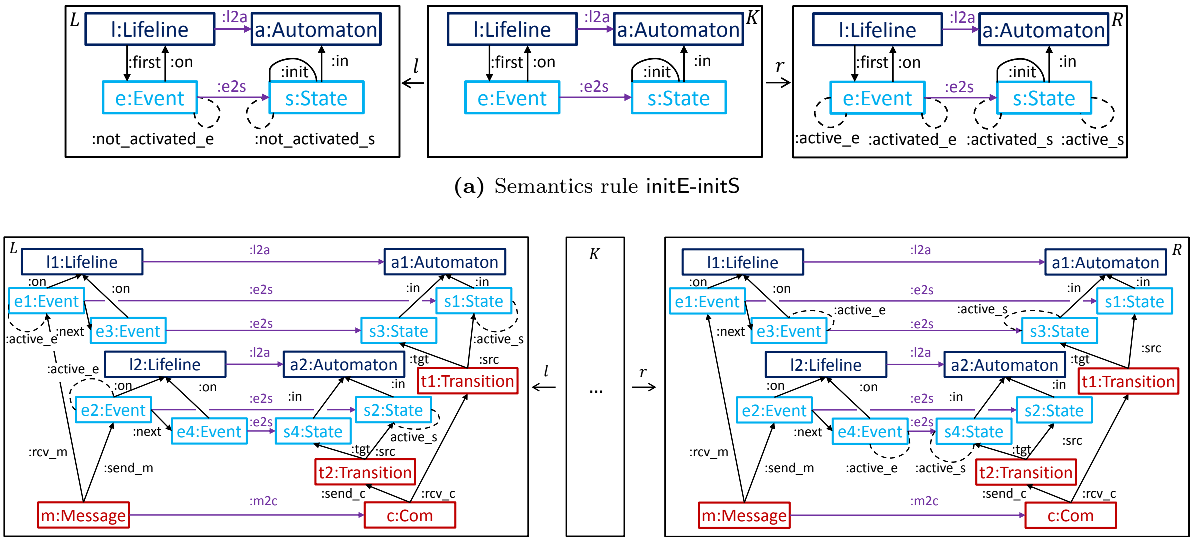


(a) Type graph TG

(b) TGG rule createInit

(c) TGG rule createMsgCom

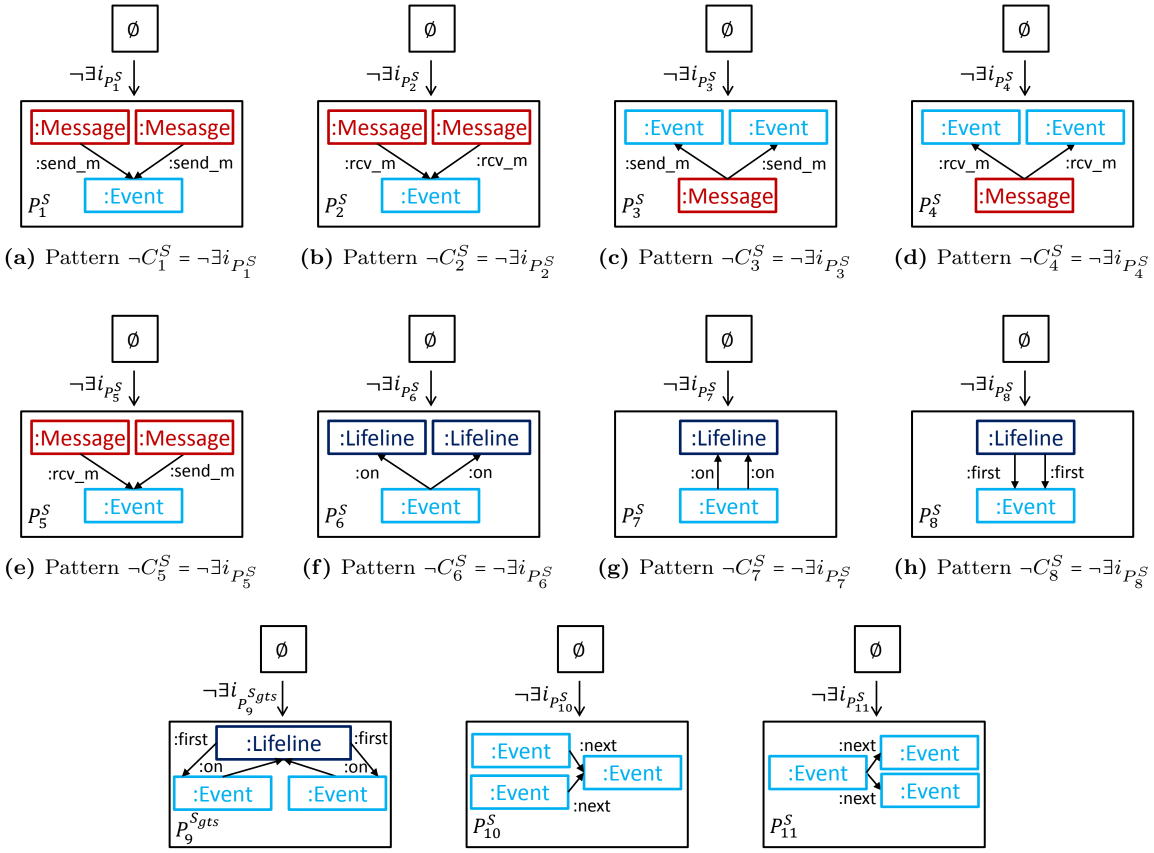
Figure C.25. – Type graph and TGG rules $\mathcal{R}$ = {createInit, createMsgCom}



(a) Semantics rule initE-initS

(b) Semantics rule send-fire

Figure C.26. – Pair rules $\mathcal{P}(l_s, l_t)^{\text{Cor}}$ = {initE-initS, send-fire}

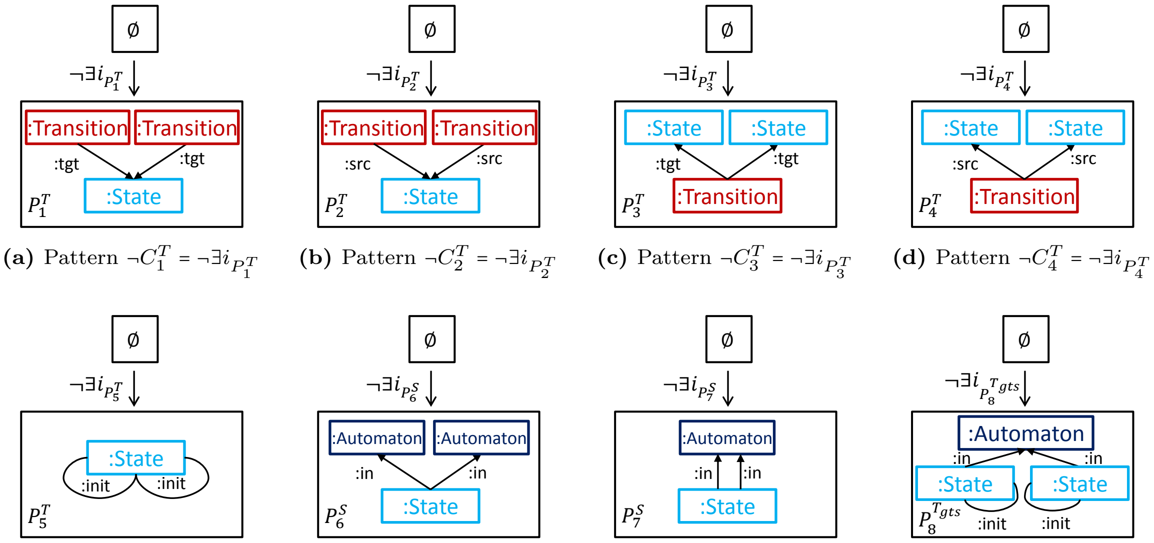


(a) Pattern $\neg C_1^S = \neg\exists i_{P_1^S}$ **(b)** Pattern $\neg C_2^S = \neg\exists i_{P_2^S}$ **(c)** Pattern $\neg C_3^S = \neg\exists i_{P_3^S}$ **(d)** Pattern $\neg C_4^S = \neg\exists i_{P_4^S}$

(e) Pattern $\neg C_5^S = \neg\exists i_{P_5^S}$ **(f)** Pattern $\neg C_6^S = \neg\exists i_{P_6^S}$ **(g)** Pattern $\neg C_7^S = \neg\exists i_{P_7^S}$ **(h)** Pattern $\neg C_8^S = \neg\exists i_{P_8^S}$

(i) Pattern $\neg C_9^S = \neg\exists i_{P_9^S}$ **(j)** Pattern $\neg C_{10}^S = \neg\exists i_{P_{10}^S}$ **(k)** Pattern $\neg C_{11}^S = \neg\exists i_{P_{11}^S}$

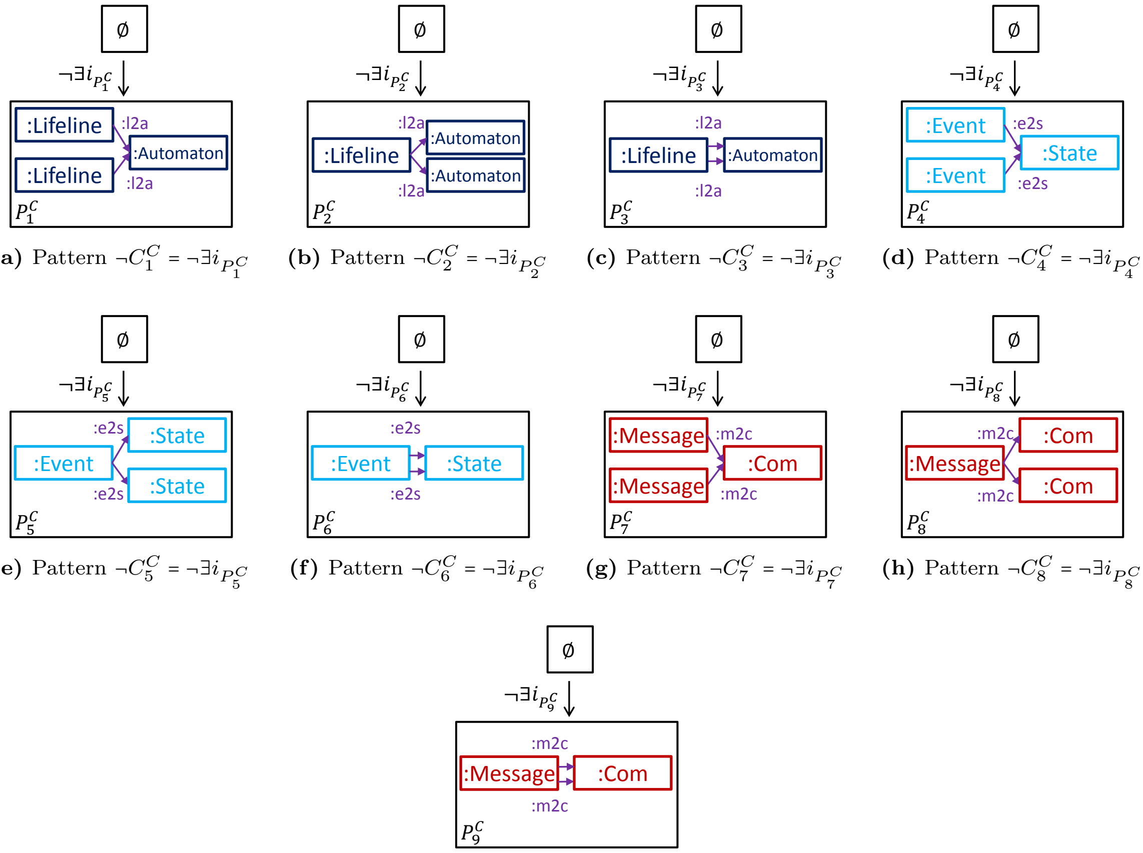
Figure C.27. – Composed guaranteed pattern $\mathcal{C}_S$



(a) Pattern $\neg C_1^T = \neg\exists i_{P_1^T}$ **(b)** Pattern $\neg C_2^T = \neg\exists i_{P_2^T}$ **(c)** Pattern $\neg C_3^T = \neg\exists i_{P_3^T}$ **(d)** Pattern $\neg C_4^T = \neg\exists i_{P_4^T}$

(e) Pattern $\neg C_5^T = \neg\exists i_{P_5^T}$ **(f)** Pattern $\neg C_6^T = \neg\exists i_{P_6^T}$ **(g)** Pattern $\neg C_7^T = \neg\exists i_{P_7^T}$ **(h)** Pattern $\neg C_8^T = \neg\exists i_{P_8^T}$

Figure C.28. – Composed guaranteed pattern $\mathcal{C}_T$



(a) Pattern $\neg C_1^C = \neg\exists i_{P_1^C}$

(b) Pattern $\neg C_2^C = \neg\exists i_{P_2^C}$

(c) Pattern $\neg C_3^C = \neg\exists i_{P_3^C}$

(d) Pattern $\neg C_4^C = \neg\exists i_{P_4^C}$

(e) Pattern $\neg C_5^C = \neg\exists i_{P_5^C}$

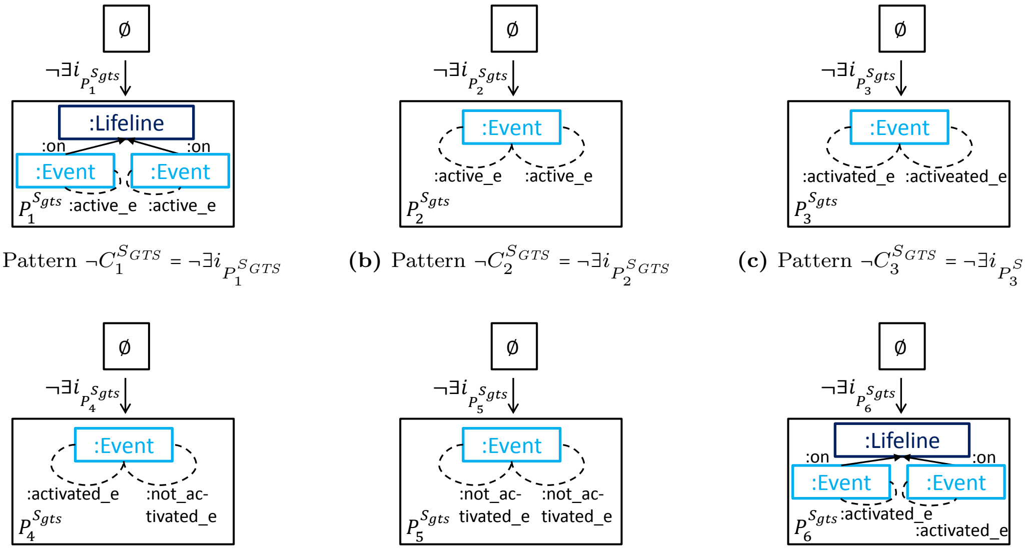
(f) Pattern $\neg C_6^C = \neg\exists i_{P_6^C}$

(g) Pattern $\neg C_7^C = \neg\exists i_{P_7^C}$

(h) Pattern $\neg C_8^C = \neg\exists i_{P_8^C}$

(i) Pattern $\neg C_9^C = \neg\exists i_{P_9^C}$

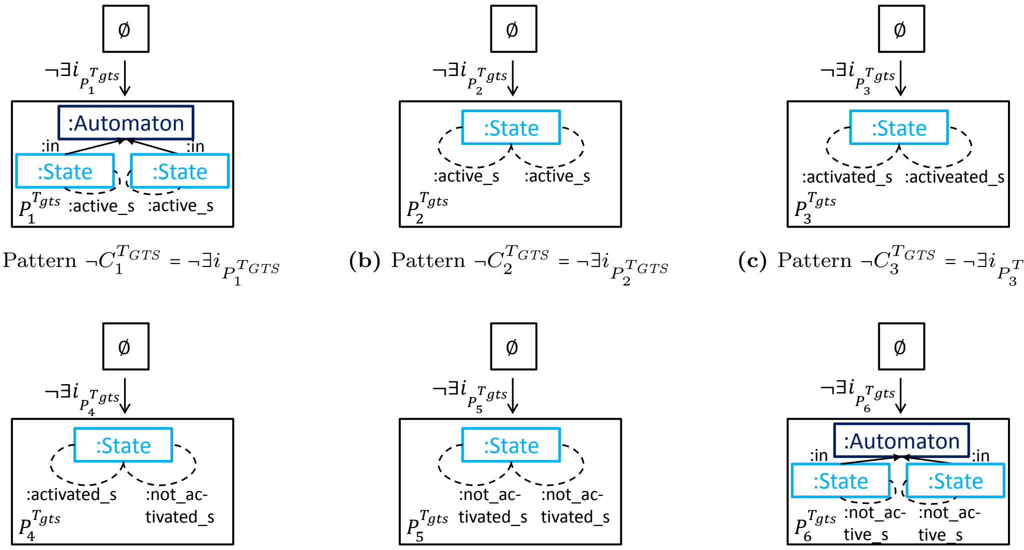
Figure C.29. – Fragments of composed guaranteed pattern $\mathcal{C}_{TGG}$



(a) Pattern $\neg C_1^{S_{GTS}} = \neg\exists i_{P_1^{S_{GTS}}}$ **(b)** Pattern $\neg C_2^{S_{GTS}} = \neg\exists i_{P_2^{S_{GTS}}}$ **(c)** Pattern $\neg C_3^{S_{GTS}} = \neg\exists i_{P_3^{S_{GTS}}}$

(d) Pattern $\neg C_4^{S_{GTS}} = \neg\exists i_{P_4^{S_{GTS}}}$ **(e)** Pattern $\neg C_5^{S_{GTS}} = \neg\exists i_{P_5^{S_{GTS}}}$ **(f)** Pattern $\neg C_6^{S_{GTS}} = \neg\exists i_{P_6^{S_{GTS}}}$

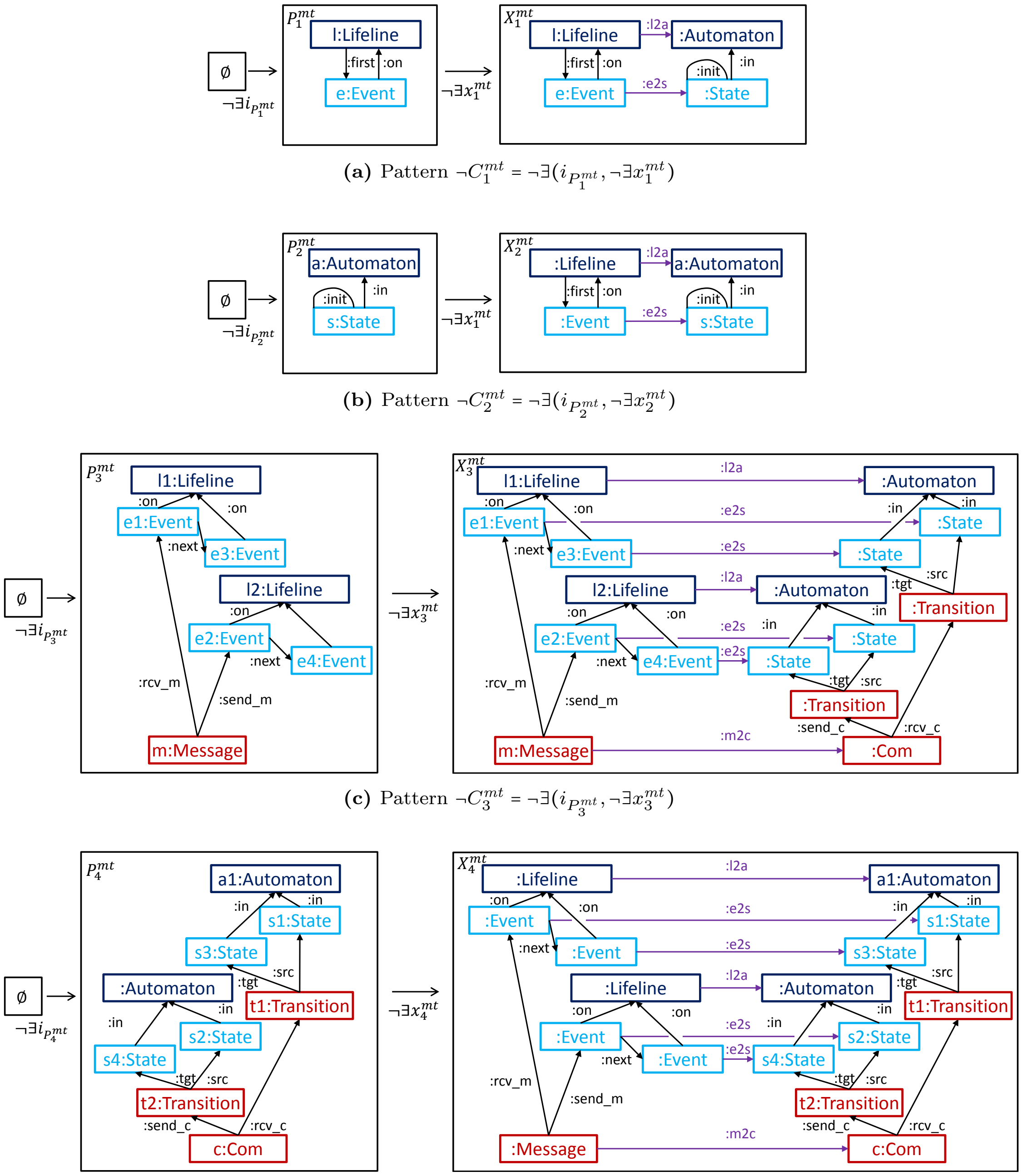
Figure C.30. – Composed pattern $\mathcal{C}_S^{GTS}$



(a) Pattern $\neg C_1^{T_{GTS}} = \neg\exists i_{P_1^{T_{GTS}}}$ **(b)** Pattern $\neg C_2^{T_{GTS}} = \neg\exists i_{P_2^{T_{GTS}}}$ **(c)** Pattern $\neg C_3^{T_{GTS}} = \neg\exists i_{P_3^{T_{GTS}}}$

(d) Pattern $\neg C_4^{T_{GTS}} = \neg\exists i_{P_4^{T_{GTS}}}$ **(e)** Pattern $\neg C_5^{T_{GTS}} = \neg\exists i_{P_5^{T_{GTS}}}$ **(f)** Pattern $\neg C_6^{T_{GTS}} = \neg\exists i_{P_6^{T_{GTS}}}$

Figure C.31. – Composed pattern $\mathcal{C}_T^{GTS}$



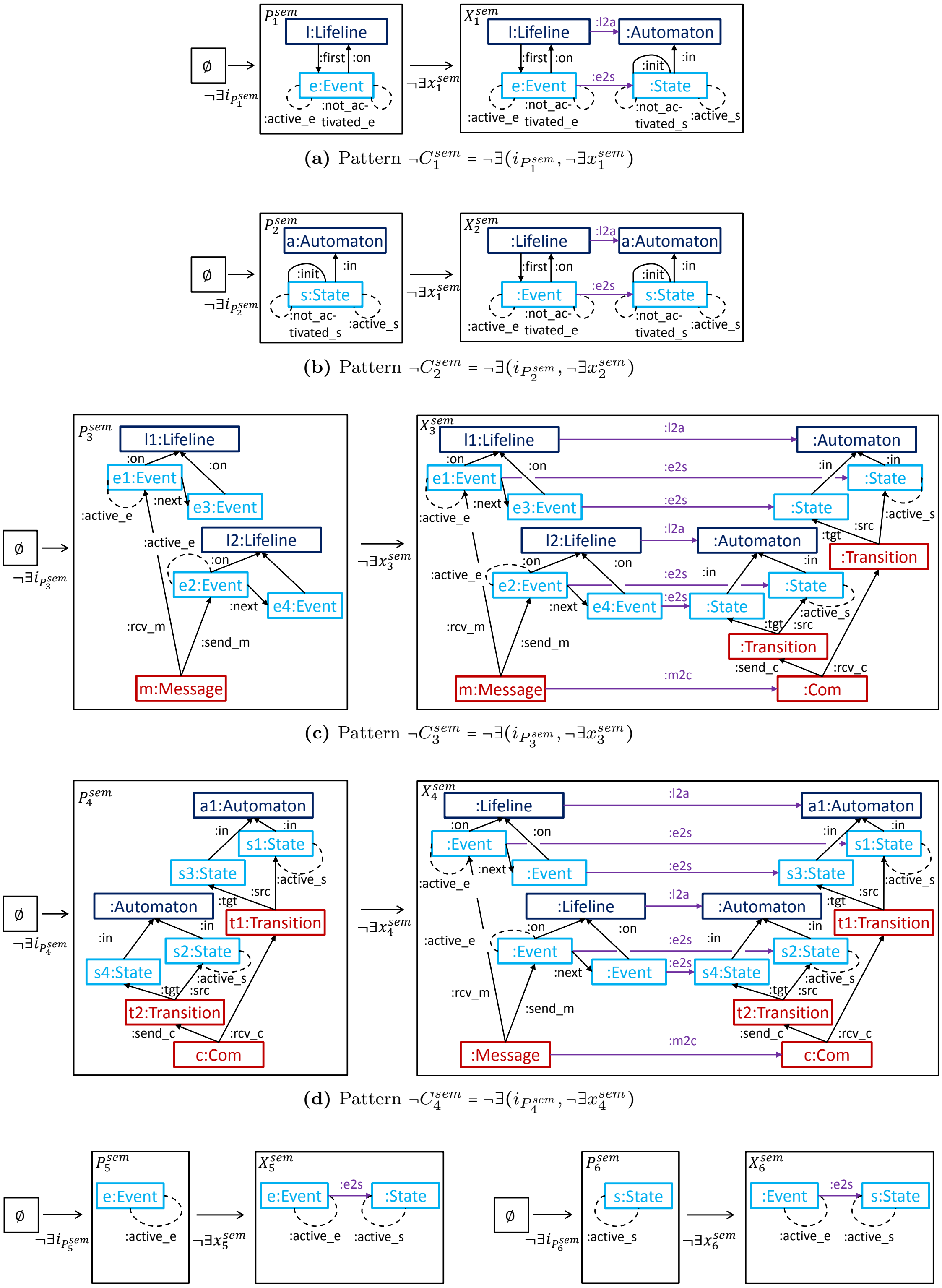
(a) Pattern $\neg C_1^{mt} = \neg\exists(i_{P_1^{mt}}, \neg\exists x_1^{mt})$

(b) Pattern $\neg C_2^{mt} = \neg\exists(i_{P_2^{mt}}, \neg\exists x_2^{mt})$

(c) Pattern $\neg C_3^{mt} = \neg\exists(i_{P_3^{mt}}, \neg\exists x_3^{mt})$

(d) Pattern $\neg C_4^{mt} = \neg\exists(i_{P_4^{mt}}, \neg\exists x_4^{mt})$

Figure C.32. – Composed pattern $\mathcal{C}_{MT}^{Cor}$



(a) Pattern $\neg C_1^{sem} = \neg\exists(i_{P_1^{sem}}, \neg\exists x_1^{sem})$

(b) Pattern $\neg C_2^{sem} = \neg\exists(i_{P_2^{sem}}, \neg\exists x_2^{sem})$

(c) Pattern $\neg C_3^{sem} = \neg\exists(i_{P_3^{sem}}, \neg\exists x_3^{sem})$

(d) Pattern $\neg C_4^{sem} = \neg\exists(i_{P_4^{sem}}, \neg\exists x_4^{sem})$

(e) Pattern $\neg C_5^{sem} = \neg\exists(i_{P_5^{sem}}, \neg\exists x_5 sem)$

(f) Pattern $\neg C_6^{sem} = \neg\exists(i_{P_6^{sem}}, \neg\exists x_6^{sem})$

Figure C.33. – Composed pattern $\mathcal{C}_{Pair}^{Cor} \wedge \mathcal{C}_{RT}$

C.2.3. Refinement – refine-trans and refine-sem

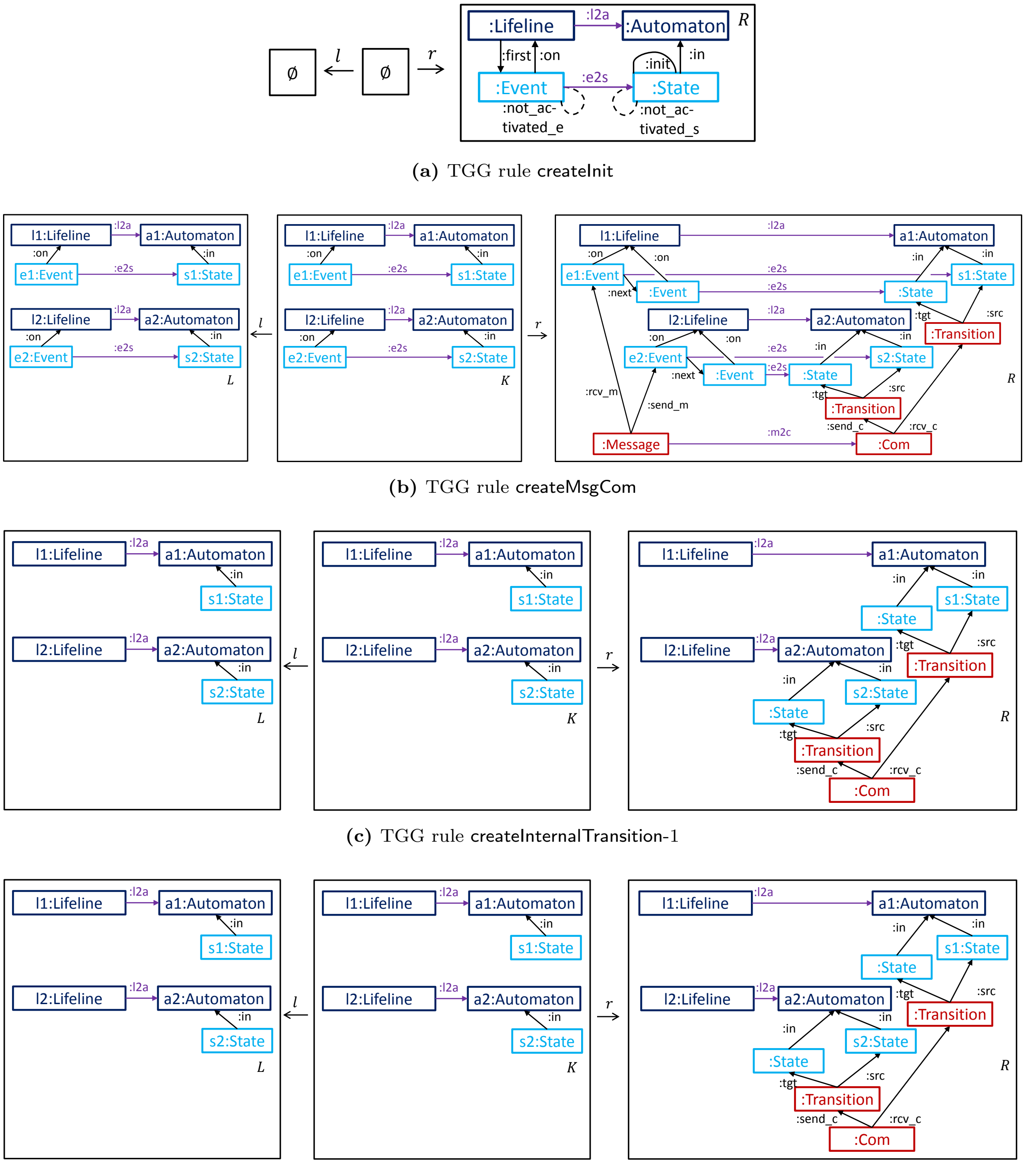
Table C.14. – Elements of refine-trans

Element		Fig.	Description
TG		C.25(a)	There are events on lifelines, connected by messages that can be received and sent. Similarly, states in a system of communicating automata are connected by communicating transitions. Correspondence elements connect corresponding elements of the source and target modeling languages.
$\mathcal{R}$	createInit	C.34(a)	Creates a lifeline and corresponding automata with an initial event and state, respectively.
	createMsgCom	C.34(b)	Creates a message that can be received and sent by events and creates two corresponding communicating transitions. The rule also creates subsequent events and states.
	createInternalTransition-1	C.34(c)	Creates internal transitions in the system of comunicating automata without an equivalence in the sequence chart.
	createInternalTransition-2	C.34(d)	Creates internal transitions in the system of comunicating automata without an equivalence in the sequence chart.
$\mathcal{F}$	$\mathcal{C}_S^{GTS}$	C.30	Runtime restrictions for semantics of source models.
	$\mathcal{C}_T^{GTS}$	C.31	Runtime restrictions for semantics of target models.
	$\mathcal{C}_{MT}^{Cor,f}$	C.32(a), C.32(c)	Model transformation constraint, which establishes correspondence properties about the static structure of source and target models.
	$\mathcal{C}_{Pair}^{Cor,f} \wedge \mathcal{C}_{RT}^{f}$	C.33(a), C.33(c), C.33(e)	Pair constraint and runtime constraint required to show the existence of a bimiulation relation.
$\mathcal{H}$	$\mathcal{C}_S$	C.27	Cardinality and type graph constraints for the source modeling language (source component of the type graph).
	$\mathcal{C}_T$	C.28(c)- C.28(h)	Cardinality and type graph constraints for the target modeling language (target component of the type graph).
	$\mathcal{C}_{TGG}$	C.29	Cardinality and type graph constraints for correspondence elements (correspondence component of the type graph).

Table C.15. – Elements of refine-sem

Element		Fig.	Description
TG		C.25(a)	There are events on lifelines, connected by messages that can be received and sent. Similarly, states in a system of communicating automata are connected by communicating transitions. Correspondence elements connect corresponding elements of the source and target modeling languages.
$\mathcal{R}$	initE-initS	C.26(a)	Initializes a sequence chart's first event and the corresponding automaton's initial state.
	send-fire	C.26(b)	Sends a message and fires the corresponding communicating transitions; also marks subsequent events and states as active.

Element		Fig.	Description
$\mathcal{F}$	$\mathcal{C}_{Pair}^{Cor,f} \wedge \mathcal{C}_{RT}^{f}$	C.33(a), C.33(c), C.33(e)	Pair constraint and runtime constraint required to show the existence of a bimiulation relation.
$\mathcal{H}$	$\mathcal{C}_S$	C.27	Cardinality and type graph constraints for the source modeling language (source component of the type graph).
	$\mathcal{C}_T$	C.28(c)- C.28(h)	Cardinality and type graph constraints for the target modeling language (target component of the type graph).
	$\mathcal{C}_{TGG}$	C.29	Cardinality and type graph constraints for correspondence elements (correspondence component of the type graph).
	$\mathcal{C}_S^{GTS}$	C.30	Runtime restrictions for semantics of source models.
	$\mathcal{C}_T^{GTS}$	C.31	Runtime restrictions for semantics of target models.
	$\mathcal{C}_{MT}^{Cor,f}$	C.32(a), C.32(c)	Model transformation constraint, which establishes correspondence properties about the static structure of source and target models.



(a) TGG rule createInit

(b) TGG rule createMsgCom

(c) TGG rule createInternalTransition-1

(d) TGG rule createInternalTransition-2

Figure C.34. – TGG rules

List of Tables