Invariant Analysis for Multi-Agent Graph Transformation Systems using k-Induction

Sven Schneider, Maria Maximova, Holger Giese

Technische Berichte Nr. 143

des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Universität Potsdam

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 143

Sven Schneider | Maria Maximova | Holger Giese

Invariant Analysis for Multi-Agent Graph Transformation Systems using k-Induction

Universitätsverlag Potsdam

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet via http://dnb.dnb.de/.

Universitätsverlag Potsdam 2022
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Phone: +49 (0)331 977 2533 / Fax: 2292
Email: verlag@uni-potsdam.de

The series **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** is edited by the professors of the Hasso Plattner Institute for Digital Engineering at the University of Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Layout: Tobias Pape
Print: docupoint GmbH Magdeburg

ISBN 978-3-86956-531-6

Also published online on the publication server of the University of Potsdam:
https://doi.org/10.25932/publishup-54585
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-545851

The analysis of behavioral models such as Graph Transformation Systems (GTSs) is of central importance in model-driven engineering. However, GTSs often result in intractably large or even infinite state spaces and may be equipped with multiple or even infinitely many start graphs. To mitigate these problems, static analysis techniques based on finite symbolic representations of sets of states or paths thereof have been devised. We focus on the technique of k-induction for establishing invariants specified using graph conditions. To this end, k-induction generates symbolic paths backwards from a symbolic state representing a violation of a candidate invariant to gather information on how that violation could have been reached possibly obtaining contradictions to assumed invariants. However, GTSs where multiple agents regularly perform actions independently from each other cannot be analyzed using this technique as of now as the independence among backward steps may prevent the gathering of relevant knowledge altogether.

In this paper, we extend k-induction to GTSs with multiple agents thereby supporting a wide range of additional GTSs. As a running example, we consider an unbounded number of shuttles driving on a large-scale track topology, which adjust their velocity to speed limits to avoid derailing. As central contribution, we develop pruning techniques based on causality and independence among backward steps and verify that k-induction remains sound under this adaptation as well as terminates in cases where it did not terminate before.

Contents

1 Introduction

The verification of formal models of dynamic systems featuring complex concurrent behavior w.r.t. formal specifications is one of the central problems in model driven engineering. However, the required expressiveness of modeling and specification formalisms that must be used for these complex dynamic systems often leads to undecidable analysis problems. For example, the formalism of GTSs considered in this paper is known to be Turing complete. Hence, fully-automatic procedures for the analysis of meaningful properties on the behavior of such GTS-based systems returning definite correct judgements cannot always terminate. Analysis becomes even more intricate when the start graph is not precisely known or when the system behavior is to be verified for a large or even infinite number of start graphs.

The technique of (forward) model checking generates the entire state space and checks this state space against the given specification. However, this technique is inapplicable when the state space is intractably large or even infinite. To mitigate this problem, large or even infinite sets of concrete states that are equivalent w.r.t. the property to be analyzed may be aggregated into symbolic states. Model checking then generates symbolic state spaces consisting of symbolic states and symbolic steps between them. However, these symbolic state spaces may still be intractably large depending on the size of the models[1] and there is usually no adequate support for multiple symbolic start states.

In backward model checking, a backward state space is generated from a set of target states derived from the specification by incrementally adding all steps leading to states that are already contained in the backward state space. For invariant properties, the target states are given by the states not satisfying the candidate invariant. As for model checking, sets of concrete states may be aggregated into symbolic states, which may also lead to a single symbolic target state. Clearly, in backward model checking, only backward paths containing exclusively reachable states are significant but during the analysis also paths containing unreachable states may be generated requiring techniques to prune such paths as soon as possible.

The technique of k-induction is a variant of bounded backward model checking for establishing state invariants. In k-induction, generated backward paths are (a) limited to length k and (b) end in a state violating the candidate invariant. Definite judgements are derived in two cases. A backward path extended to a start state leads to candidate invariant refutation and the candidate invariant is confirmed when no backward path of length k is derivable.

[1] Approaches such as CEGAR [4] also aim at minimizing symbolic state spaces.

In this paper, we extend earlier work on k-induction from [6, 19] by solving the following open problem. When the system under analysis features concurrency such as in a multi-agent context, backward steps may be independent as k backward steps may be performed by k different agents that may be logically/spatially apart. In that case, the k backward steps do not accumulate knowledge on why the violating graph could be reached preventing the derivation of a definite judgement. This problem can even occur when every target state contains a single agent since backward steps can still introduce further agents. To solve this problem, we introduce several novel GTS-specific pruning techniques. Firstly, we prune backward paths in which the last added step does not depend on the already accumulated knowledge. This *causality pruning* avoids the inclusion of steps of unrelated agents in a backward path. Secondly, we prune states containing an agent that is permanently blocked from further backward steps. This *evolution pruning* (assuming that agents existed in the start graph or are created in some step) is required when all backward steps of a certain agent have been pruned by some other pruning technique (while other agents are still able to perform backward steps). Thirdly, when a state is removed in evolution pruning, we propagate this state prunability forward across backward steps until the blocked agent has an alternative backward step. This *evolution-dependency pruning* is, in conjunction with our explicit handling of independent steps, able to prune also other backward paths (with common suffix) where independent steps of other agents are interleaved differently. For these three novel pruning techniques, we ensure that they do not affect the correctness of derived judgements and that our approach presented here is a conservative extension in the sense that it terminates whenever the single-agent approach terminated before.[2]

As a running example, we consider an unbounded number of shuttles driving on a large-scale track topology, which avoid collisions with each other. As a candidate invariant to be confirmed, shuttles in fast driving mode should not drive across construction sites to avoid derailing. To ensure this candidate invariant, warnings are installed at a certain distance in front of construction sites. Agents in this running example are the shuttles and backward steps can be performed by different shuttles on the track topology. However, only the steps of the single shuttle violating the speed limit at a construction site as well as (possibly) the steps of shuttles that forced the shuttle to navigate to that construction site are in fact relevant to the analysis. Any other steps (possibly of shuttles far away on the considered track topology) should not be considered during analysis. Hence, the novel pruning techniques are designed to focus our attention on the relevant steps of relevant agents only.

Invariant analysis for GTSs has been intensively studied. Besides the approach from [19], which is restricted to single-agent GTSs, earlier approaches for establishing invariants for GTSs lack a formal foundation such as [2] or are restricted to

[2] Intuitively, GTSs have no built-in support for different agents as opposed to other non-flat formalisms (such as e.g. process calculi) where a multi-agent system is lazily constructed using a parallel composition operation where interaction steps between agents are then resolved at runtime. For such different formalisms, causality is much easier to analyze but it is one of the many strengths of GTSs that agents can interact in complex patterns not restricted by the formalism at hand.

k-induction for $k = 1$ such as [7] or to syntactically limited nested conditions such as [6]. Moreover, tools such as Groove [12], Henshin [11], and AutoGraph [18] can be used for invariant analysis if the considered GTSs induce small finite state spaces. However, there are some approaches that also support invariant analysis for infinite state spaces. For example, the tool Augur2 [1] abstracts GTSs by Petri nets but imposes restrictions on graph transformation rules thereby limiting expressiveness. Moreover, static analysis of programs for GTSs w.r.t. pre/post conditions has been developed in [16] and [17]. Finally, an approach for the verification of invariants (similar to k-induction) is considered in [23] where graphs are abstracted by single so-called shape graphs, which have limited expressiveness compared to the nested graph conditions used in this work.

The representation of causality and the focus on causally connected steps during analysis is important in various domains. For example, for Petri nets where tokens can be understood as agents, event structures and causal/occurrence nets have been used extensively to represent causality in a given run (see e.g. [15, 21, 22]). Similarly, causality-based analysis can also be understood as cone of influence-based analysis [3] where events are derived to be insignificant when they are logically/spatially disconnected from considered events.

This paper is structured as follows. In chapter 2, we recapitulate the technique of k-induction based on labeled transition systems. In chapter 3, we recall preliminaries on graph transformation and introduce our running example. In chapter 4, we present an abstraction of GTSs to symbolic states and steps. In chapter 5, we extend existing notions capturing causality and compatibility among steps to the employed symbolic representation. In chapter 6, we discuss the k-induction procedure with the novel pruning techniques relying on causality and fairness among multiple agents in the GTS. Finally, in chapter 7, we close the paper with a conclusion and an outlook on future work.

2 Labeled Transition Systems and k-Induction

A *Labeled Transition System* (*LTS*) $\mathcal{L} = (Q, Z : Q \rightarrow \mathbf{B}, L, R \subseteq Q \times L \times Q)$ consists of a set of states Q, a state predicate Z identifying start states in Q, a set L of step labels, and a binary step relation R on Q where each step has a step label from L. An LTS $\mathcal{L}$ represents a state space and induces paths $\tilde{\pi} \in \Pi(\mathcal{L})$ traversing through its states. We write $\mathcal{L}_1 \subseteq \mathcal{L}_2$ and $\mathcal{L}_1 \cup \mathcal{L}_2$ for their componentwise containment and union, respectively.

A state predicate $P : Q \rightarrow \mathbf{B}$ is an *invariant* of $\mathcal{L}$ when P is satisfied by all states reachable from start states. A *shortest violation* of an invariant is given by a path $\tilde{\pi}$ of length n traversing through states s_i when (a) $\tilde{\pi}$ starts in a start state and never revisits a start state (i.e., $Z(s_i)$ iff $i = 0$) and (b) $\tilde{\pi}$ ends in a violating state and never traverses another violating state (i.e., $\neg P(s_i)$ iff $i = n$).

The k-induction procedure attempts to decide whether a shortest violation for a candidate invariant P exists. For shortest violations, in iteration $0 \leq i \leq k$ the paths of length i that may be suffixes of shortest violations are generated. That is, in iteration $i = 0$, all paths of length 0 consisting only of states q satisfying $\neg P(q)$ are generated. In iterations $i > 0$, each path $\tilde{\pi}$ of length $i - 1$ starting in state q is extended to paths $\tilde{\pi}'$ of length i by prepending all backward steps $(q', a, q) \in R$ such that $P(q')$ is satisfied. The k-induction procedure (a) rejects the candidate invariant P when in some iteration a path starting in a start state is generated, (b) confirms the candidate invariant P when in some iteration no path is derived, and (c) terminates without definite judgement when in the last iteration $i = k$ some path is generated.

Pruning techniques restrict the set of generated paths in each iteration to a relevant subset and only the retained paths are then considered for the the abortion criteria (a)–(c). While the additional computation that is required for pruning can be costly, pruning can speed up the subsequent iterations by reducing the number of paths to be considered in the next iteration. More importantly, pruning may prevent the generation of paths of length k, which lead to an indefinite judgement. For example, when $A : Q \rightarrow \mathbf{B}$ is an *assumed invariant* (either established in an earlier application of the same or another technique or assumed without verification), all paths in which some state q satisfies $\neg A(q)$ are pruned as in [6, 19] attempting to limit constructed paths to reachable states. Further pruning techniques introduced later on are designed specifically for the case of GTSs taking the content of states and the nature of steps among them into account.[1]

[1] The computational trade-off between pruning costs and costs for continued analysis of retained paths will play out differently for each example but, due to the usually exponential number of paths of a certain length, already the rather simple pruning technique based on assumed invariants was highly successful in [6, 19] where it was also required to establish a definite judgement at all.

3 Graph Transformation and Running Example

Our approach generalizes to the setting of $\mathcal{M}$-adhesive categories and $\mathcal{M}$-adhesive transformation systems with nested application conditions as introduced in [10]. Nevertheless, to simplify our presentation, we consider the $\mathcal{M}$-adhesive category of typed directed graphs (short graphs) using the fixed type graph TG from Figure 3.1a (see [8, 9, 10] for a detailed introduction). In visualizations of graphs such as Figure 3.1b, types of nodes are indicated by their names (i.e., S_i and T_i are nodes of type Shuttle and Track) whereas we only use the type names for edges. We denote the empty graph by $\emptyset$, monomorphisms (monos) by $f : H \hookrightarrow H'$, and the initial morphism for a graph H by $\mathrm{i}(H) : \emptyset \rightarrowtail H$. Moreover, a graph is finite when it has finitely many nodes and edges and a set S of morphisms with common codomain X is jointly epimorphic, if morphisms $g, h : X \rightarrow Y$ are equal when $\forall f \in S.\ g \circ f = h \circ f$ holds.

In our running example, we consider an unbounded number of shuttles driving on a large-scale track topology where subsequent tracks are connected using *next* edges (see again TG in Figure 3.1a and the example graph in Figure 3.1b). Each shuttle either drives fast or slow (as marked using *fast* or *slow* loops). Shuttles approaching track-forks (i.e., a track with two successor tracks) decide non-deterministically between the two successor tracks. Certain track-forks consist of a regular successor track and an emergency exit successor track (marked using an *ee* loop) to be used only to avoid collisions with shuttles on the regular successor track. Construction sites may be located on tracks (marked using *cs* loops) and, to inform shuttles about construction sites ahead, warnings are installed four tracks ahead of them (marked using *warn* edges instead of *next* edges). To exclude the possibility of shuttles derailing, analysis should confirm the candidate invariant **P** stating that shuttles never drive fast on construction sites. Assumed invariants are used to rule out track topologies with undesired characteristics such as missing *warn* edges. We model this shuttle scenario using a GTS with rules featuring application conditions as well as assumed and candidate invariants all given by (nested) Graph Conditions (GCs). For this purpose, we now recall GCs and GTSs in our notation.

The graph logic GL from [10] allows for the specification of sets of graphs and monos using GCs. Intuitively, for a host graph G, a GC over a finite subgraph H of G given by a mono $m : H \hookrightarrow G$ states the presence (or absence) of graph elements in G based on m. In particular, the GC $\exists(f : H \hookrightarrow H', \phi')$ requires that m must be extendable to a match $m' : H' \hookrightarrow G$ of a larger subgraph H' where the nested sub-GC ϕ' restricts m'. The combination of propositional operators and the nesting of exis-

tential quantifications results in an expressiveness equivalent to first-order logic on graphs [5].

Definition 1 (Graph Conditions (GCs))**.** If H is a finite graph, then ϕ *is a graph condition* (*GC*) *over* H, written $\phi \in \mathsf{GC}(H)$, if an item applies.

- $\phi = \neg\phi'$ and $\phi' \in \mathsf{GC}(H)$.
- $\phi = \vee(\phi_1, \ldots, \phi_n)$ and $\{\phi_1, \ldots, \phi_n\} \subseteq \mathsf{GC}(H)$.
- $\phi = \exists(f : H \hookrightarrow H', \phi')$ and $\phi' \in \mathsf{GC}(H')$.

Note that the empty disjunction $\vee()$ serves as a base case not requiring the prior existence of GCs. We obtain the derived operators *false* $\bot$, *true* $\top$, *conjunction* $\wedge(\phi_1, \ldots, \phi_n)$, and *universal quantification* $\forall(f, \phi)$ in the expected way.

We now define the two satisfaction relations of GL capturing (a) when a mono $m : H \hookrightarrow G$ into a host graph G satisfies a GC over H and (b) when a graph G satisfies a GC over the empty graph $\emptyset$.

Definition 2 (Satisfaction of GCs)**.** A mono $m : H \hookrightarrow G$ satisfies a GC ϕ over H, written $m \models \phi$, if an item applies.

- $\phi = \neg\phi'$ and $\neg(m \models \phi')$.
- $\phi = \vee(\phi_1, \ldots, \phi_n)$ and $\exists 1 \leq i \leq n.\ m \models \phi_i$.
- $\phi = \exists(f : H \hookrightarrow H', \phi')$ and $\exists m' : H' \hookrightarrow G.\ m' \circ f = m \wedge m' \models \phi'$.

A graph G satisfies a GC ϕ over the empty graph $\emptyset$, written $G \models \phi$, if the (unique) initial morphism $\mathrm{i}(G) : \emptyset \hookrightarrow G$ satisfies ϕ.

For our running example, (a) the GC ϕ_{AI} from Figure 3.1h expresses the assumed invariant stating that there is always a warning preceding each construction site[1], (b) the GC ϕ_{CI} from Figure 3.1i expresses the candidate invariant **P** stating that there is no fast shuttle at a track with a construction site, and (c) the GC ϕ_{SC} from Figure 3.1g expresses that there is no fast shuttle already in the critical section between a warning and a construction site. Note that in visualizations of GCs, we represent monos $f : H \hookrightarrow H'$ in quantifications by only visualizing the smallest subgraph of H' containing $H' - f(H)$.

We rely on the operation shift from e.g. [10] for shifting a GC ϕ over a graph H across a mono $g : H \hookrightarrow H'$ resulting in a GC $\mathsf{shift}(g, \phi)$ over H'. The following fact states that GC shifting essentially expresses partial GC satisfaction checking for a morphism decomposition $f \circ g$.

Fact 1 (Operation shift [10])**.** $f \models \mathsf{shift}(g, \phi)$ iff $f \circ g \models \phi$

GTSs with multiple start graphs are now defined by specifying these start graphs using a GC over the empty graph. We employ the Double Pushout (DPO) approach to graph transformation with nested application conditions (see [8, 9, 10] for details) in which rules contain two morphisms $\ell : K \hookrightarrow L$ and $r : K \hookrightarrow R$ describing the removal of the elements in $L - \ell(K)$ and the addition of elements in $R - r(K)$ as well as a left-hand side (nested) application condition given by a GC over L to be satisfied by the match morphism.

[1] To ease the presentation, we omit further assumed invariants excluding graphs with duplicate *next* edges or tracks with more than two successor/predecessor tracks.

Definition 3 (Graph Transformation System (GTS))**.** A pair $S = (\phi_0, P)$ is a *graph transformation system* (*GTS*), if ϕ_0 is a GC over the empty graph $\emptyset$ and P is a finite set of graph transformation rules (short rules) of the form $\rho = (\ell : K \hookrightarrow L, r : K \hookrightarrow R, \phi)$ where L, K, and R are finite and ϕ is a GC over L.

If G, G' are graphs, $\sigma = (\rho, m : L \hookrightarrow G, n : R \hookrightarrow G')$ is a step label containing a rule $\rho = (\ell : K \hookrightarrow L, r : K \hookrightarrow R, \phi)$ of S, a match m,[2] and a comatch n, the DPO diagram in Figure 3.2a exists, and $m \models \phi$, then $G \Rightarrow_\sigma G'$ is a (GT) step of the LTS $\mathcal{L}_{graphs}$ induced by the GTS S. Also, the notion of derived rules $\mathsf{drule}(\sigma) = (f, g, \mathsf{shift}(m, \phi))$ captures the transformation span of the step and the instantiated application condition.

For our running example, we employ the GTS $S = (\phi_{\mathsf{SC}} \wedge \phi_{\mathsf{CI}}, \{\rho_{\mathsf{drive}}, \rho_{\mathsf{driveEE}}, \rho_{\mathsf{warnS}}, \rho_{\mathsf{warnF}}\})$ using the GCs and rules from Figure 3.1. For each rule, we use an integrated notation in which L, K, and R are given in a single graph where graph elements marked with $\ominus$ are from $L - \ell(K)$, graph elements marked with $\oplus$ are from $R - r(K)$, and where all other graph elements are in K. The application condition of each rule is given on the left side of the $\triangleright$ symbol. The rule ρ_{drive} states that a shuttle can advance to a next track T_2 when no other shuttle is on T_2 and when T_2 is not marked to be an emergency exit. The rule ρ_{driveEE} states that a shuttle can advance to a next track T_2 marked to be an emergency exit when the regular successor track T_3 is occupied by another shuttle. The rule ρ_{warnS} states that a slow shuttle can advance to a next track T_2 passing by a warning when no other shuttle is on T_2. Finally, the rule ρ_{warnF} states that a fast shuttle can slow down and advance to a next track T_2 passing by a warning when no other shuttle is on T_2.

To accumulate the knowledge captured in application conditions and the candidate invariant over steps of a backward path, we employ the operation L from e.g. [10] for shifting a GC ϕ' over a graph R across a rule $\rho = (\ell : K \hookrightarrow L, r : K \hookrightarrow R, \phi)$ resulting in a GC $\mathsf{L}(\rho, \phi')$ over L. The following fact states that the operation L translates post-conditions of steps into equivalent pre-conditions.

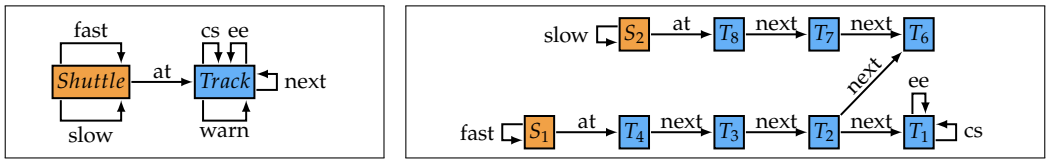
Fact 2 (Operation L [10])**.** $G \Rightarrow_{\rho,m,n} G'$ implies $(m \models \mathsf{L}(\rho, \phi')$ iff $n \models \phi')$.

For our running example, we expect the k-induction procedure to confirm the candidate invariant ϕ_{CI} for $k \geq 4$ realizing that a fast shuttle at a construction site must have passed by a warning 4 steps earlier due to the assumed invariant ϕ_{AI}, which ensures that the shuttle drives slowly onto the construction site later on.[3] When applying the k-induction procedure, we start with the minimal graph G_{vio} representing a violation (see the graph used in ϕ_{CI} in Figure 3.1i). To extend a given backward path from G to G_{vio} by prepending a backward step using a certain rule, we first extend G to a graph E by adding graph elements to then be able to apply the rule backwards to E (as discussed in more detail in the next section based on a symbolic representation of states and steps). Consider the graph G_{ex} in Figure 3.1b,

[2] Note that our approach extends to the usage of general match morphisms.

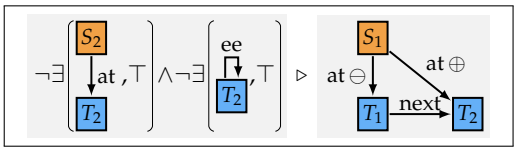
[3] The candidate invariant ϕ_{CI} could also be violated because (a) it is not satisfied by all start graphs (which is excluded since $\phi_{\mathsf{SC}} \wedge \phi_{\mathsf{CI}}$ captures the start graphs of the GTS), (b) a slow shuttle becomes a fast shuttle between a warning and a construction site (for which no rule exists in the GTS), and (c) a pair of a warning and a construction site could wrap a fast shuttle at runtime (for which no rule exists in the GTS).

which can be reached using this iterative backward extension from G_{vio} by a path of length 5 (see Figure A.1a). Since the relevant shuttle S_1 has no further enabled *backward* step from G_{ex} according to the rules of the GTS (because fast shuttles cannot advance backwards over *warn* edges), any path leading to G_{ex} and any other path that varies by containing additional/fewer/differently ordered independent steps can be pruned (as discussed in more detail in chapter 6). For example, the similar path (see Figure A.1b) where the shuttle S_2 has only been moved backwards to T_6 is pruned as well. Hence, with such additional pruning techniques, we mitigate the problem that the relevant shuttle S_1 does not move backwards in every backward step of every path. Instead, it is sufficient that S_1 is being moved backwards three times in some path. Still, all interleavings of backward steps must be generated (since, for arbitrary GTSs, it cannot be foreseen which interleaving results in a prunable path later on) but pruning one of these paths can result in the pruning of many further paths.

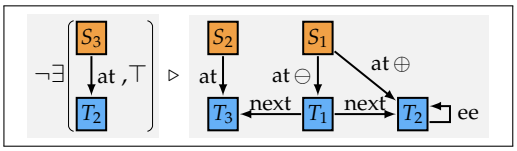


(**a**) Type graph TG.

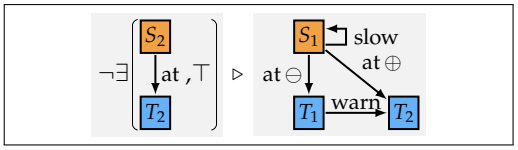
(**b**) Example graph G_{ex}.



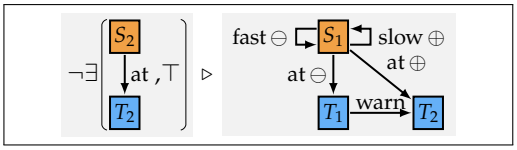
(**c**) Rule ρ_{drive}: shuttle moves to free track without emergency exit.



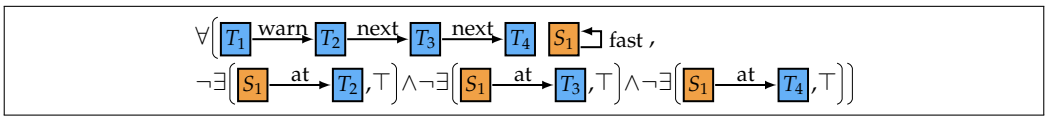
(**d**) Rule ρ_{driveEE}: shuttle moves to free emergency exit track.



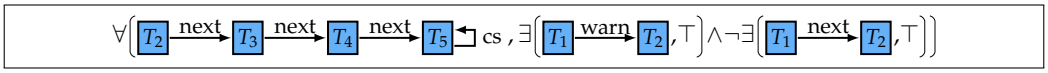
(**e**) Rule ρ_{warnS}: slow shuttle passes warning to free track.



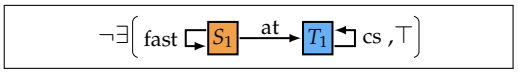
(**f**) Rule ρ_{warnF}: fast shuttle passes warning to free track and slows down.



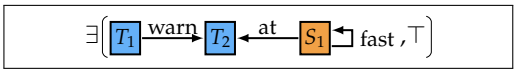
(**g**) Start condition ϕ_{SC}: there is no fast shuttle on the three tracks following a warning.



(**h**) Assumed invariant ϕ_{AI}: there is a warning precisely four tracks ahead of any construction site on any branch of tracks leading to the construction site.

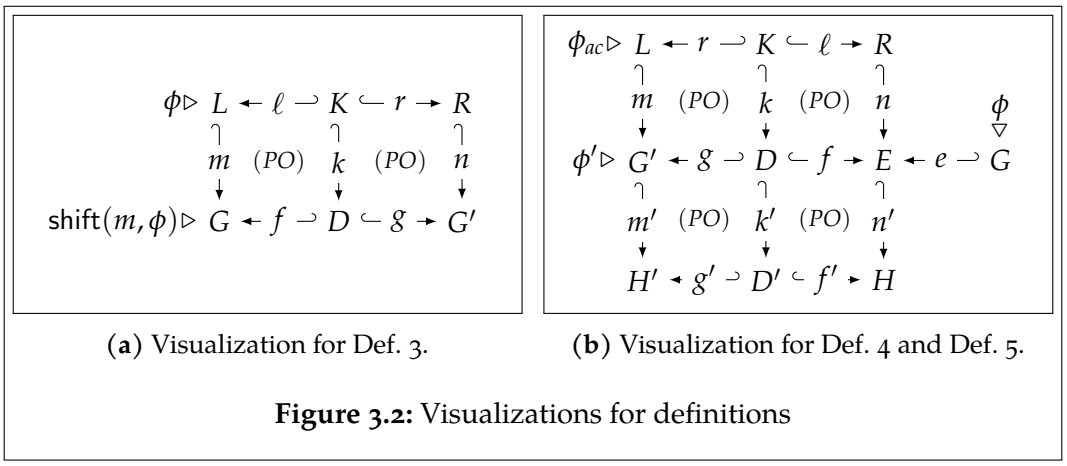


(**i**) Candidate invariant $\phi_{\mathsf{CI}} = \neg\exists(\mathrm{i}(G_{vio}), \top)$: no fast shuttles on construction sites.



(**j**) Blocked agent GC ϕ_{BA}: fast shuttles cannot move backwards across warnings.

Figure 3.1: Running example



(**a**) Visualization for Def. 3.

(**b**) Visualization for Def. 4 and Def. 5.

Figure 3.2: Visualizations for definitions

4 Symbolic States and Steps

Following [19], concrete states of a GTS are given by graphs and its symbolic states are given by pairs (G, ϕ) of a graph G and a GC ϕ over G. A symbolic state (G, ϕ) represents all graphs H for which some $m : G \hookrightarrow H$ satisfies ϕ.

This symbolic representation extends to GT steps and symbolic steps and paths thereof. To obtain a backward step from a state (G, ϕ) (cf. Figure 3.2b), (i) G is overlapped with the right-hand side graph R of some rule ρ where the overlapping consists of the comatch n of the backward step and the embedding morphism e and (ii) the GC ϕ and the application condition ϕ_{ac} of the rule are shifted to the resulting symbolic state (G', ϕ'). As discussed before, further graph elements are added using e as required for the k-induction procedure in which we start with (usually very small) graphs representing violations and then accumulate additional context *also* in terms of additional graph elements.

Definition 4 (Symbolic Step)**.** If (G', ϕ') and (G, ϕ) are symbolic states, $\rho = (\ell : K \hookrightarrow L, r : K \hookrightarrow R, \phi_{ac})$ is a rule, $\sigma = (\rho, m : L \hookrightarrow G', n : R \hookrightarrow E)$ is a step label, $G' \Rightarrow_\sigma E$ is a DPO step, $e : G \hookrightarrow E$ is a mono, e and n are jointly epimorphic, and $\phi' = \mathsf{L}(\mathsf{drule}(\sigma), \mathsf{shift}(e, \phi)) \wedge \mathsf{shift}(m, \phi_{ac})$, then $(G', \phi') \blacktriangleright_{\sigma,e} (G, \phi)$ is a symbolic step of the LTS $\mathcal{L}_{symb}$ induced by the GTS S (see Figure 3.2b).

To obtain concrete paths $\hat{\pi}$ represented by a symbolic path π, the implicit requirements given by the GCs in symbolic states and the incremental context extensions via monos e are resolved. This entails a forward propagation of additional graph elements resulting in a consistent perspective throughout all graphs traversed in $\hat{\pi}$. However, making these additional graph elements explicit may change satisfaction judgements for application conditions and assumed or candidate invariants implying that a symbolic path may represent no concrete path relevant in the context of k-induction or even no concrete path at all. Since some pruning techniques require that we are able to operate on the symbolic step relation, we only concretize symbolic paths using forward propagation that may represent concrete paths being shortest violations.

Definition 5 (Concretization of Symbolic Path)**.** A concrete path $\hat{\pi}$ is a concretization of a symbolic path π with first state (G', ϕ') for a mono $m' : G' \hookrightarrow H'$ satisfying ϕ', written $\hat{\pi} \in \mathsf{refine}(\pi, m')$, if an item applies.

- $\pi = (G', \phi')$ and $\hat{\pi} = H'$.
- $\pi = (G', \phi') \cdot \sigma \cdot e \cdot (G, \phi) \cdot \pi'$, $\sigma = (\rho, m, n)$, $\sigma' = (\rho, m' \circ m, n' \circ n)$, $H' \Rightarrow_{\sigma'} H$, $\hat{\pi}' \in \mathsf{refine}((G, \phi) \cdot \pi', n' \circ e)$, and $\hat{\pi} = H' \cdot \sigma' \cdot \hat{\pi}'$ (see Figure 3.2b).

The symbolic representation given by symbolic paths is complete in the sense of the following lemma stating that the concrete paths of a GTS correspond to the concretizations of all symbolic paths.

Lemma 1 (Full Coverage). $\Pi(\mathcal{L}_{graphs}) = \bigcup\{\mathsf{refine}(\pi, m') \mid \pi \in \Pi(\mathcal{L}_{symb})\}$
See page 32 for the proof of this lemma.

For our running example, the violating symbolic state used for the symbolic paths of length 0 during k-induction is $(G_{vio}, \top)$ (see Figure 3.1i). Note that we implicitly rewrite symbolic states (G, ϕ) into symbolic states (G', ϕ') using the symbolic model generation technique from [20] to accumulate all positive requirements of G and ϕ in the graph G' and to store the remaining negative requirements (stating how G' cannot be extended) in ϕ'. Without this technique, k-induction would be limited to candidate invariants of the form $\neg\exists(\mathrm{i}(G), \top)$ and graph patterns required by positive application conditions would not be explicitly contained in the graph and could therefore not be overlapped leading to indefinite judgements in some cases. However, if multiple states (G', ϕ') are obtained using this rewriting, we would perform k-induction for each of these states separately. For the running example, $(G_{vio}, \top)$ is obtained by rewriting $(\emptyset, \neg\neg\exists(\mathrm{i}(G_{vio}), \top))$ using this technique.

5 Causality and Independence in GTS

According to [8, p. 8] in the context of GTSs, causal independence of rule applications allows for their execution in arbitrary order.

In the general setting of an LTS $\mathcal{L}$, considering Figure 5.1a, (a) the two parallel steps with source s_3 (to s_1 and s_2), (b) the two parallel steps with target s_0 (from s_1 and s_2), (c) the two sequential steps traversing through s_1 (from s_3 and to s_0), or (d) the two sequential steps traversing through s_2 (from s_3 and to s_0) are independent iff the respective remaining two steps exist resulting in the square given in Figure 5.1a (which we represent by $((s_3, b_2, s_1), (s_1, a_1, s_0), (s_3, a_2, s_2), (s_2, b_1, s_0)) \in \mathsf{SQ}(\mathcal{L})$) where, for $x \in \{a, b\}$, the labels x_1 and x_2 are required to be equivalent in an LTS specific sense in each case (a)–(d). Clearly, in such an obtained square, each pair of sequential steps is sequentially independent and each pair of parallel steps (with common source/target) is parallel independent.

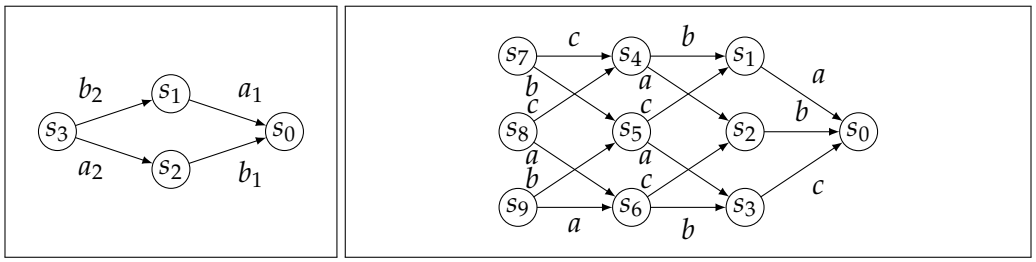
Subsequently, we call two successive steps *causally connected* when they are sequentially dependent and, correspondingly, two alternative steps *incompatible* when they are parallel dependent. In the context of k-induction where steps are derived backwards, we primarily consider parallel independence for steps with common target graph.

Sequential and parallel independence for common source graphs have been formalized for GTSs in [10] (see also Definition 6 on page 32 for their formal definitions). The reverse notion of parallel independence for common target graphs is derived as expected essentially relying on the fact that GT steps can be reversed by applying the reversed rule. The Local Church-Rosser Theorem (see [10, Theorem 4.7]) provides the results corresponding to the discussion for LTSs from above. Technically, for concrete GT steps, for $x \in \{a, b\}$, two step labels σ_{x_1} and σ_{x_2} must then use the same rule and must match essentially the same graph elements.[1] Moreover, for symbolic steps, for $x \in \{a, b\}$, we additionally require that the step labels σ_{x_1}, e_{x_1} and σ_{x_2}, e_{x_2} state the same extensions using e_{x_1} and e_{x_2}.[2]

We use the operation linearize to obtain all linearizations for a given set of parallel steps with common target. For example, given the two parallel steps with target s_0 in Figure 5.1a, linearize constructs the two further backward steps and the square given in Figure 5.1a when the two steps are parallel independent and no further backward steps and no square otherwise. In general, for a given LTS $\mathcal{L}$ and a subset $\delta \subseteq Q \times L \times Q$ of size $n \geq 0$ of parallel steps of $\mathcal{L}$ with common target, $\mathsf{linearize}(\delta) = (\overline{sq}, \delta')$ generates the set $\overline{sq} \subseteq \mathsf{SQ}(\mathcal{L})$ of all squares that can be constructed by rearranging

[1] The considered GT steps must preserve the matched graph elements and thereby explain how one match is propagated over a GT step resulting in the other match.

[2] Similarly to the requirement on matches, which must essentially match the same graph elements, the extension monos must extend the graphs with the same graph elements up to the propagation along the considered symbolic steps.



(**a**) Square of four steps. (**b**) Multiple squares obtained by iterative linearization.

Figure 5.1: Linearization of parallel independent backward steps

those parallel steps into corresponding sequences of length at most n and a set δ' of all generated steps including δ. More precisely, linearize iteratively constructs a square for each pair of distinct parallel independent steps with common target (considering for this the steps from δ and all steps generated already).[3] For the cases of $n = 0$ and $n = 1$ no additional steps are generated. For the cases of $n = 2$ and $n = 3$, Figure 5.1a and Figure 5.1b depict the maximal set δ' of resulting steps that may be generated when all pairs of distinct parallel steps are parallel independent throughout the application of linearize (note that we omit in Figure 5.1b the differentiation between different a_i, b_i, and c_i steps for improved readability).

When some pair of steps with common target is not parallel independent (which is often the case), fewer squares and steps are generated.

[3]For concrete GT steps, we rely on [10, Theorem 4.7] to obtain a construction procedure for the operation linearize. Also, this construction procedure extends to the case of symbolic steps as the additional GCs in symbolic states are extended precisely by the application conditions of the two involved rules in exchanged order only.

6 Causality-Based k-Induction and Pruning Techniques

We now present our adaptation of the k-induction procedure from chapter 2 defined on an arbitrary LTS $\mathcal{L}_c$ but apply this procedure later on only to the LTS $\mathcal{L}_{symb}$ induced by the symbolic step relation from chapter 4. Hence, $\mathcal{L}_c$ is only assumed to be available in terms of its step relation and a method for identifying start states as well as states satisfying the assumed invariant.[1] Hereby, we rely on the notion of parallel independence of steps with common target and linearizations of such steps resulting in sequences of sequentially independent steps as introduced in the previous section. The paths derived within this procedure consist then of steps from $\mathcal{L}_c$ and are given in the procedure by a partial LTS $\mathcal{L}_p$ contained in the complete LTS $\mathcal{L}_c$. The k-induction procedure has a start state $\mathfrak{q}_0$ and modifies this state up to k times using a single step of type $\mathcal{Q} \longrightarrow \mathcal{Q}$ as explained subsequently in more detail.

States of k-induction: The traversed states $\mathfrak{q} \in \mathcal{Q}$ are of the form $(\mathcal{L}_c, \mathcal{L}_p, N, \overline{sq})$ where $\mathcal{L}_c = (Q_c, Z_c, L_c, R_c)$ is the complete LTS as discussed above, $\mathcal{L}_p = (Q_p, Z_p, L_p, R_p) \subseteq \mathcal{L}_c$ is a partial LTS contained in $\mathcal{L}_c$ recording the steps derived so far, $N \subseteq Q_p$ is the subset of states to be considered next, and $\overline{sq} \subseteq \mathsf{SQ}(\mathcal{L}_c)$ records the derived squares of independent steps.

Start state of k-induction: For a given complete LTS $\mathcal{L}_c$ and a state $q_0 \in Q_c$ violating the candidate invariant from which backward paths are constructed, the start state $\mathfrak{q}_0$ of k-induction is given by $\mathfrak{q}_0 = (\mathcal{L}_c, (\{q_0\}, \emptyset, \emptyset, \emptyset), \{q_0\}, \emptyset)$.

Single step of k-induction: The single step of k-induction executes (a) the operation $\mathsf{extend} : \mathcal{Q} \longrightarrow \mathcal{Q}$ generating additional steps with target in N, extending the LTS $\mathcal{L}_p$ by these steps and all further steps obtained using linearization, and then (b) the operation $\mathsf{prune} : \mathcal{Q} \longrightarrow \mathcal{Q}$ applying pruning techniques. The operation extend first derives the set $\delta = \{(q, a, q') \in R_c \mid q' \in N\}$ of all backward steps with target in N and generates all linearizations $\mathsf{linearize}(\delta) = (\overline{sq}_{ext}, \delta_{ext})$ of these steps.[2] The operation extend then returns $\mathsf{extend}(\mathfrak{q}) = \mathfrak{q}' = (\mathcal{L}_c, \mathcal{L}'_p, N', \overline{sq}')$ where $\mathcal{L}'_p = \mathcal{L}_p \cup \mathcal{L}_{ext}$ is obtained by merging the previous partial LTS with the extension $\mathcal{L}_{ext} =$

[1] A symbolic state (G, ϕ) satisfies the start state condition ϕ_{SC} (or analogously an assumed invariant ϕ_{AI}) iff $\phi \wedge \phi_{\mathsf{SC}}$ is satisfiable. The model generation procedure from [20] implemented in the tool AutoGraph [18] can be used to check GCs for satisfiability (if it returns unknown, the problem must be delegated to the user for ϕ_{SC} and satisfiability may be assumed for ϕ_{AI}). If $\phi \wedge \neg\phi_{\mathsf{SC}}$ (or, analogously, $\phi \wedge \neg\phi_{\mathsf{AI}}$) is also satisfiable, not every concretization of paths $(G, \phi) \cdot \pi$ will be a violation. This source of overapproximation can be eliminated using splitting of states as in [19].

[2] Note that, due to linearization, R_p may already contain some of the steps derived here. By implicitly comparing steps derived here to those in R_p, we ensure to not derive isomorphic copies of steps. Also, two distinct parallel independent steps do not need to be linearized if not both steps are already contained in R_p.

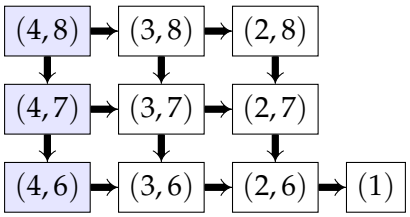


Figure 6.1: Fragment of backward state space constructed for running example. We abbreviate symbolic states by only providing a tuple of the track numbers on which shuttles are located. See Figure 3.1b for the graph part of state (4,8)

$(Q'_p, \{q \mapsto Z_c(q) \mid q \in Q'_p\}, \{a \mid (q, a, q') \in \delta\}, \delta_{ext})$ containing all steps derived in the current iteration using the set of all states $Q'_p = \{q \mid (q, a, q') \in \delta_{ext}\}$ derived in the current iteration, $N' = \{q \mid (q, a, q') \in \delta\}$ contains all predecessor states of those in N, and $\overline{sq}' = \overline{sq} \cup \overline{sq}_{ext}$ additionally includes all squares derived in the current iteration. The operation prune is then applied to $\mathfrak{q}'$ and discussed separately below.

For our running example, consider Figure 6.1 where, initially in state (1), there is a single (fast) shuttle S_1 located on track T_1. A second shuttle S_2 is then added onto track T_6 in the first backward step to (2,6). When reaching the state (4,8), of which the graph part is given in Figure 3.1b, the shuttles S_1 and S_2 moved backwards three and two times, respectively. See also Figure A.1a for this backward path from $(4,8)$ to (1) and an additional backward path in Figure A.1b from $(4,6)$ to (1), which is also included in abbreviated form in Figure 6.1. The pruning of state $(4,6)$ in Figure A.1b due to the blocked agent (given by the shuttle S_1) leads to the pruning of also the states $(3,6)$, $(2,6)$, and (1) in Figure A.1b and consequently also the path in Figure A.1a.

Termination condition of k-induction: The k-induction procedure applies the single step up to k times on the start state $\mathfrak{q}_0$. When a state is derived with $N = \emptyset$, the procedure concludes satisfaction of the candidate invariant. When a state is derived with Z_p mapping some state q to $\top$, the procedure concludes non-satisfaction of the candidate invariant and returns $(\mathcal{L}_p, q)$ as a counterexample. When the single step has been applied k times and none of the previous two cases applies, the procedure returns an indefinite judgement.

GTS-specific pruning: For the GTS setting where $\mathcal{L}_c = \mathcal{L}_{symb}$ as discussed above, we now present five pruning techniques (where the first two have been used already in [6, 19]), which are used to remove certain states (and all steps depending on these states) recorded in the partial LTS $\mathcal{L}'_p$.

For *assumed invariant pruning*, we remove all states not satisfying the assumed invariant ϕ_{AI} as in prior work on GTS k-induction. For our running example, when moving the shuttle S_1 backwards from (4,6), a *next* edge is added leading to track T_4, which is forbidden by the assumed invariant ϕ_{AI} from Figure 3.1h. Hence, this backward step of that shuttle is pruned.

For *realizability pruning*, we first determine states q that are identified to be start states via $Z_p(q) = \top$. Since each such state q represents a violating path leading to

the refutation of the candidate invariant at the end of the iteration, we attempt to exclude false positives where each symbolic path π in $\mathcal{L}_p$ from q to the violating state q_0 cannot be concretized to a GTS path according to Definition 5. For this purpose, for $q = (G, \phi)$, we use the model generation procedure from [20] to generate extensions $m : G \hookrightarrow G'$ satisfying $\phi \wedge \phi_{\mathsf{SC}} \wedge \phi_{\mathsf{AI}}$. We then attempt to concretize some symbolic path π from q to q_0 to a concrete path $\hat{\pi}$ using m. If some $\hat{\pi}$ is obtained representing a shortest violation, the k-induction procedure terminates after this iteration refuting the candidate invariant. If the model generation procedure does not terminate, q may be a false positive and the k-induction procedure terminates with an indefinite judgement. However, if both cases do not apply, q is removed from $\mathcal{L}'_p$.

Certainly, *any* derived state q may not allow for a concretization along the same lines. However, not checking each such state for realizability along the same lines may only lead to indefinite judgements and there is a trade-off between the cost for realizability pruning and the cost of exponentially more backward extensions leading to q to be generated and analyzed.

For *causality pruning*, a state q' is pruned when there is some symbolic backward step $q' \Rightarrow_{(\rho,m,n),e} q$ where n and e have non-overlapping images. We thereby ensure that the number of weakly connected components[3] of the graph under transformation does not increase over backward steps. For our running example, we prune states where further shuttles are added that are structurally not connected to the subgraph originating from the start state. Note that further shuttles can still be included as for the graph in Figure 3.1b where the shuttle S_2 has been added according to the rule ρ_{driveEE} used in the first backward step.

For *evolution pruning*, a state q is pruned when it contains an agent (given in our running example by shuttles) for which permanent blockage is detected. Note that, as explained in chapter 1, the inability of some agent to partake in a backward step does not preclude the ability of some other agent to partake in a backward step. Hence, when not removing such states, irrelevant steps of additional agents may prolong analysis or even prevent definite judgements. Also note that an agent is in general allowed to be blocked forever when it reaches its local configuration in a start graph of the GTS allowing other agents to perform backward steps to jointly reach a start graph. Since GTSs are Turing complete, no precise identification of such agents can be achieved and, to preclude the derivation of incorrect judgements, we must underapproximate the set of such agents. Technically, we attempt to identify all agents in states q that will unexpectedly never again be able to partake in a backward step using an additional blocked agent GC ϕ_{BA}. Such a blocked agent GC is (a finite disjunction of GCs) of the form $\exists(\mathrm{i}(H), \top)$ where H represents a minimal pattern containing a blocked agent. For our running example, see Figure 3.1j for the GC ϕ_{BA} capturing a fast shuttle (i.e., an agent) that is blocked by not being able to move backwards across a *warn* edge. To maintain soundness of k-induction, we can verify the blocked agent GC ϕ_{BA} by checking that there is no symbolic backward step from $(H, \top)$ preventing that any further backward steps from q can reach a state where

[3] Two nodes n_1 and n_2 of a graph G are in a common weakly connected component (given by a set of nodes of G) of G iff there is a sequence of the edges of G from n_1 to n_2 where edges may be traversed in either direction.

the matched agent can partake in a backward step. A state $q = (G, \phi)$ is then pruned using the blocked agent GC ϕ_{BA} when $\exists(\mathrm{i}(G), \phi) \wedge \phi_{\mathsf{AI}} \wedge \phi_{\mathsf{BA}}$ is satisfiable. For our running example, the shuttle S_1 is blocked according to the GC ϕ_{BA} in the states (4,6), (4,7), and (4,8) (marked blue in Figure 6.1), which are therefore pruned.

For *evolution-dependency pruning*, we extend the state-based evolution pruning to a step- and square-based pruning technique propagating the information about blocked agents forward across steps. In particular, given a step (q', a, q) where q' was pruned (due to a blocked agent), q is also pruned unless there is a backward step (q'', b, q) to a non-pruned state q'' that is parallel dependent to (q', a, q). The step (q'', b, q) then potentially represents an alternative backward path not leading to a blocked agent.[4] However, only relying on the notion of parallel independence, considering steps from a global perspective and not tracking which agents actually participated in the two backward steps, can lead to an underapproximation of the steps that can be pruned, potentially leading to avoidable indefinite returned judgements.[5] That is, backward steps of two distinct agents can be parallel dependent, which would then not allow to propagate the knowledge of one of them being blocked forwards. Constructing explicitly the squares in our backward state space generation procedure is essential for dissecting alternative backward steps. The forward propagation of prunability thereby allows to prune states and hence also all other paths traversing through these additionally pruned states where different step interleavings (of other agents) are executed (hence assuming that the blocked agent would be treated unfairly in all these other paths).

The usage of squares in *k*-induction supports evolution-dependency pruning since pruning a state also prunes all paths traversing through it, which would not be the case when we would construct a set of (disconnected) backward sequences or a tree (or forest) of backward steps. Moreover, minimizing the size of the state space representation using squares reduces the number of states for which blocked agents must be detected and from which evolution-dependency pruning must be performed. Also, when only constructing backward sequences instead, there would e.g. in our running example be a backward path not moving the initially given shuttle S_1 backwards to a situation where that shuttle would be blocked. Hence, employing a directed acyclic graph given by the square-based compressed backward state space, we can easily detect states occurring in different backward paths and thereby do not need to treat fairness among different agents beyond generating the backward state space using breadth-first search.

For our running example, the pruning of the state (4,6) and the non-existence of a backward step parallel dependent to the step from (4,6) to (3,6) leads to the pruning of the state (3,6) as well. Analogously, the states (2,6) and then (1) are also pruned leaving an empty state space, which leads to termination and candidate invariant confirmation at the end of the iteration.

Finally, we state that the presented *k*-induction procedure is sound and at least as complete as the previous variants from [6, 19].

[4] Parallel independent backward steps are always performed by different agents.

[5] This pruning technique can be refined by attributing agents to steps to then determine prunable states with greater precision complicating forward propagation.

Theorem 1 (Soundness of k-Induction)**.** For a given GTS S, a candidate invariant ϕ_{CI}, an assumed invariant ϕ_{AI}, and a blocked agent GC ϕ_{BA}, the k-induction procedure confirms/refutes ϕ_{CI} only if ϕ_{CI} is an invariant/is no invariant. Also, it returns such a definite judgement whenever the k-induction procedure from [6, 19] without the novel pruning techniques and the use of causality and independence did.

Proof (*Sketch*). Extending [6, 19], we only need to ensure that the *novel* pruning techniques never prune states/paths that would otherwise be extended to shortest violations (the pre-existing assumed invariant pruning and realizability pruning do not need to be reexamined here). Causality pruning only removes steps where a disconnected agent is introduced: these steps can never help in gathering knowledge about the past of the actors involved in the violation and, moreover, the inclusion of such disconnected agents can always be delayed to later steps where they are then connected to a part of the current graph. The validity of the blocked agent GC ϕ_{BA} ensures that evolution pruning only prunes states containing an agent permanently blocked precluding the reachability of a start graph of the GTS. Evolution-dependency pruning then only prunes states/paths from which that agent unavoidably reaches such a blocking situation lacking alternative backward steps. □

7 Conclusion and Future Work

We extended the *k*-induction procedure from [6, 19] to support the verification of state invariants also for multi-agent GTSs. The presented extension relies on novel pruning techniques determining generated backward paths that cannot be extended to paths capturing a violation of the candidate invariant. It only returns sound judgements on candidate invariants, succeeds when the prior versions in [6, 19] did, and succeeds for additional multi-agent GTSs.

In the future, we will extend our approach to Probabilistic Timed Graph Transformation Systems (PTGTSs) [13] in which dependencies among agents are also induced by the use of clocks (as in timed automata). This additional coupling among agents will complicate our analysis but will also reduce the number of possible backward paths to be constructed. Moreover, we will extend our prior implementations on *k*-induction to the presented approach and will evaluate the expected performance gain when restricting backward steps to a fixed underlying static topology fragment as in [14].

Bibliography

[1] *Augur 2*. Universität Duisburg-Essen. 2008. URL: http://www.ti.inf.uni-due.de/en/research/tools/augur2.

[2] B. Becker and H. Giese. "On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles". In: *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*. IEEE Computer Society, 2008, pages 203–210. ISBN: 978-0-7695-3132-8. DOI: 10.1109/ISORC.2008.13.

[3] S. Berezin, S. V. A. Campos, and E. M. Clarke. "Compositional Reasoning in Model Checking". In: *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*. Edited by W. P. de Roever, H. Langmaack, and A. Pnueli. Volume 1536. Lecture Notes in Computer Science. Springer, 1997, pages 81–102. DOI: 10.1007/3-540-49213-5_4.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Edited by E. A. Emerson and A. P. Sistla. Volume 1855. Lecture Notes in Computer Science. Springer, 2000, pages 154–169. ISBN: 3-540-67770-4. DOI: 10.1007/10722167_15.

[5] B. Courcelle. "The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic". In: *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. Edited by G. Rozenberg. World Scientific, 1997, pages 313–400. ISBN: 9810228848.

[6] J. Dyck. "Verification of Graph Transformation Systems with k-Inductive Invariants". PhD thesis. University of Potsdam, Hasso Plattner Institute, Potsdam, Germany, 2020. DOI: 10.25932/publishup-44274.

[7] J. Dyck and H. Giese. "k-Inductive Invariant Checking for Graph Transformation Systems". In: *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*. Edited by J. de Lara and D. Plump. Volume 10373. Lecture Notes in Computer Science. Springer, 2017, pages 142–158. ISBN: 978-3-319-61469-4. DOI: 10.1007/978-3-319-61470-0_9.

[8] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

[9] H. Ehrig, C. Ermel, U. Golas, and F. Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015. ISBN: 978-3-662-47979-7. DOI: 10.1007/978-3-662-47980-3.

[10] H. Ehrig, U. Golas, A. Habel, L. Lambers, and F. Orejas. "$\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation". In: *Mathematical Structures in Computer Science* 24.4 (2014). DOI: 10.1017/S0960129512000357.

[11] *EMF Henshin*. The Eclipse Foundation. 2013. URL: http://www.eclipse.org/modeling/emft/henshin.

[12] *Graphs for Object-Oriented Verification (GROOVE)*. University of Twente. 2011. URL: http://groove.cs.utwente.nl.

[13] M. Maximova, H. Giese, and C. Krause. "Probabilistic timed graph transformation systems". In: *J. Log. Algebr. Meth. Program.* 101 (2018), pages 110–131. DOI: 10.1016/j.jlamp.2018.09.003.

[14] M. Maximova, S. Schneider, and H. Giese. "Compositional Analysis of Probabilistic Timed Graph Transformation Systems". In: *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Edited by E. Guerra and M. Stoelinga. Volume 12649. Lecture Notes in Computer Science. Springer, 2021, pages 196–217. DOI: 10.1007/978-3-030-71500-7_10.

[15] M. Nielsen, G. D. Plotkin, and G. Winskel. "Petri Nets, Event Structures and Domains, Part I". In: *Theor. Comput. Sci.* 13 (1981), pages 85–108. DOI: 10.1016/0304-3975(81)90112-2.

[16] K.-H. Pennemann. "Development of correct graph transformation systems". URN: urn:nbn:de:gbv:715-oops-9483. PhD thesis. University of Oldenburg, Germany, 2009.

[17] C. M. Poskitt and D. Plump. "Verifying Monadic Second-Order Properties of Graph Programs". In: *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings*. Edited by H. Giese and B. König. Volume 8571. Lecture Notes in Computer Science. Springer, 2014, pages 33–48. ISBN: 978-3-319-09107-5. DOI: 10.1007/978-3-319-09108-2_3.

[18] S. Schneider. *AutoGraph*. URL: https://github.com/schneider-sven/AutoGraph.

[19] S. Schneider, J. Dyck, and H. Giese. "Formal Verification of Invariants for Attributed Graph Transformation Systems Based on Nested Attributed Graph Conditions". In: *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*. Edited by F. Gadducci and T. Kehrer. Volume 12150. Lecture Notes in Computer Science. Springer, 2020, pages 257–275. DOI: 10.1007/978-3-030-51372-6_15.

[20] S. Schneider, L. Lambers, and F. Orejas. "Automated reasoning for attributed graph properties". In: *STTT* 20.6 (2018), pages 705–737. DOI: 10.1007/s10009-018-0496-3.

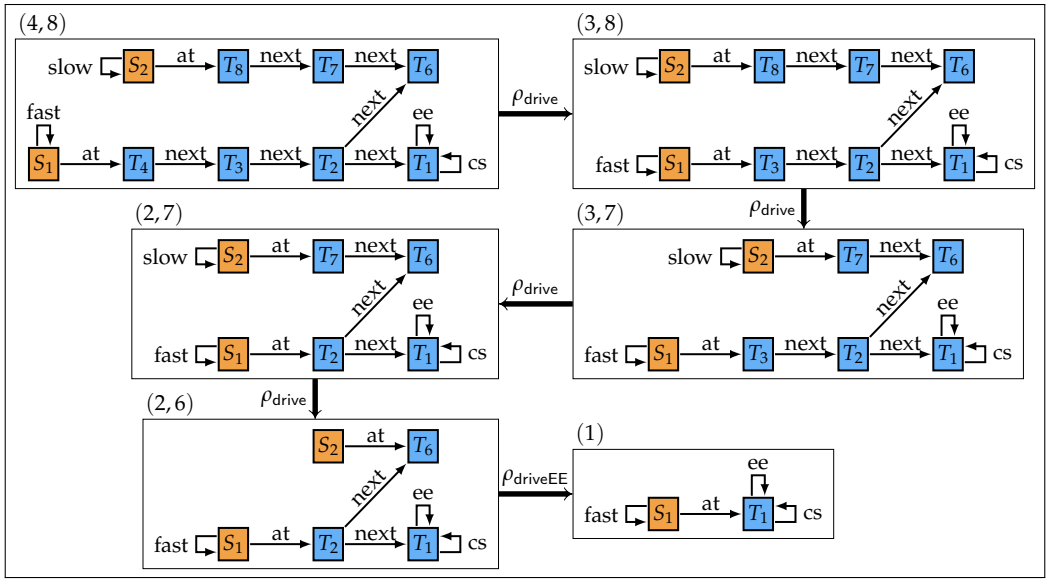
[21] E. Smith. "On net systems generated by process foldings". In: *Advances in Petri Nets 1991, Papers from the 11th International Conference on Applications and Theory of Petri Nets, Paris, France, June 1990*. Edited by G. Rozenberg. Volume 524. Lecture Notes in Computer Science. Springer, 1990, pages 253–276. DOI: 10.1007/BFb0019978.

[22] E. Smith. "On the Border of Causality: Contact and Confusion". In: *Theor. Comput. Sci.* 153.1&2 (1996), pages 245–270. DOI: 10.1016/0304-3975(95)00123-9.

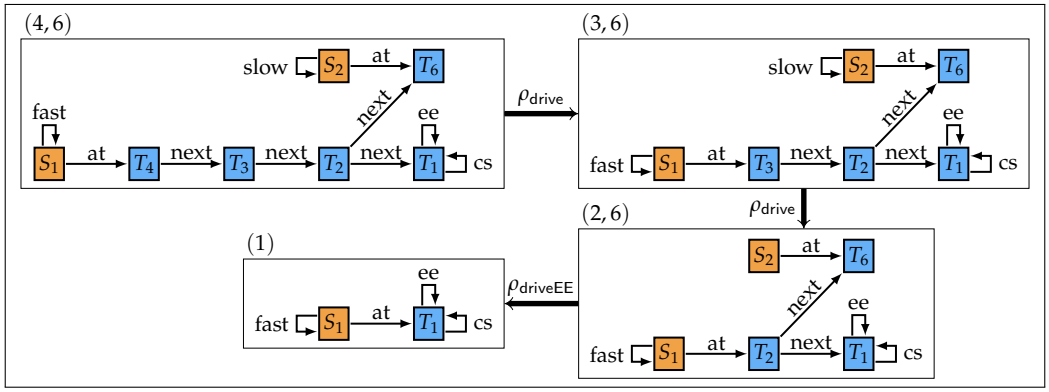
[23] D. Steenken. "Verification of infinite-state graph transformation systems via abstraction". PhD thesis. University of Paderborn, 2015.

A Two Backward Paths

In this appendix, we provide two backward paths in Figure A.1, which have been given in quite abbreviated form already in Figure 6.1. The pruning of state $s0$ in Figure A.1b due to the blocked agent (given by the shuttle S_1) leads to the pruning of also the states $s1$–$s3$ in Figure A.1b and consequently also the path in Figure A.1a.



(**a**) Longest backward path in Figure 6.1 leading to a blocked agent.



(**b**) Shortest backward path in Figure 6.1 leading to a blocked agent.

Figure A.1: Two backward paths

B Additional Results and Proofs

In this appendix, we give more technical details for some used concepts and notions as well as provide proofs for stated lemmas.

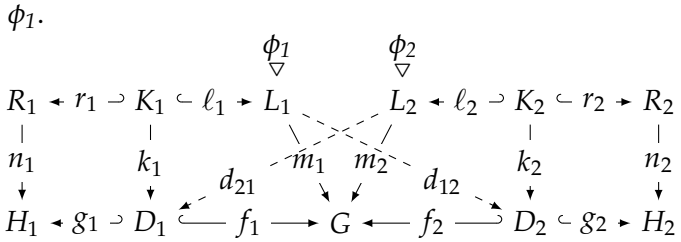
Proof for Lemma 1, p. 19: Full Coverage. We show equality by mutual inclusion.

- Direction $\subseteq$:
  We show that every concrete path $\hat{\pi} \in \Pi(\mathcal{L}_{graphs})$ starting in a graph G is in $\mathsf{refine}(\pi, \mathrm{id}(G))$ where $\pi \in \Pi(\mathcal{L}_{symb})$ is equal to $\hat{\pi}$ but with all e monos being identities and all additional GCs being $\top$. We proceed by induction.
  - Fix $\hat{\pi} = G'$, $m' = \mathrm{id}(G')$, and $\pi = (G', \top)$. Then $\hat{\pi} \in \mathsf{refine}(\pi, \mathrm{id}(G'))$ since $\mathrm{id}(G') \models \top$.
  - Fix $\hat{\pi} = G' \cdot \sigma \cdot G \cdot \hat{\pi}'$, $m' = \mathrm{id}(G')$, and $\pi = (G', \phi') \cdot \sigma \cdot \mathrm{id}(G) \cdot (G, \phi) \cdot \pi'$ where $G \cdot \hat{\pi}' \in \mathsf{refine}((G, \phi) \cdot \pi', \mathrm{id}(G))$ by induction. Also fix $\sigma = (\rho, m, n)$. Then $\hat{\pi} \in \mathsf{refine}(\pi, \mathrm{id}(G))$ since $\mathrm{id}(G) \models \top$ and (for well-formedness of π, we have $G' \Rightarrow_{\sigma} G$ directly from $\hat{\pi}$ being a path) $G' \Rightarrow_{\sigma'} G$ with $\sigma' = (\rho, \mathrm{id}(G') \circ m, \mathrm{id}(G) \circ n) = \sigma$ from the well-definedness of π.
- Direction $\supseteq$:
  We show that every concretization $\hat{\pi}$ of a symbolic path π can be generated accordingly directly from the GTS. We proceed by induction.
  - Fix $\pi = (G', \phi')$, m' with $m' \models \phi'$, and $\hat{\pi} = G'$. Then $\hat{\pi} \in \Pi(\mathcal{L}_{graphs})$.
  - Fix $\pi = (G', \phi') \cdot \sigma \cdot e \cdot (G, \phi) \cdot \pi'$, $\sigma = (\rho, m, n)$, $m' : G' \hookrightarrow H'$ with $m' \models \phi'$, $\hat{\pi} = H' \cdot \sigma' \cdot \hat{\pi}'$, $\sigma' = (\rho, m'_2 \circ m, n'_2 \circ n)$, and $G' \Rightarrow_{\sigma'} G$ where $\hat{\pi}' \in \Pi(\mathcal{L}_{graphs}) \cap \mathsf{refine}((G, \phi) \cdot \pi', n' \circ e)$ by induction. Then $\hat{\pi} \in \Pi(\mathcal{L}_{graphs})$ since $G' \Rightarrow_{\sigma'} G$.

□

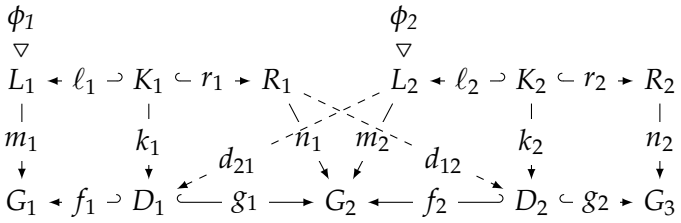
According to [10], the formal definition of parallel and sequential independence of GT steps with Application Conditions (ACs) is as follows.

Definition 6 (Parallel and Sequential Independence [10, Def. 4.3])**.** A pair of GT steps $G_1 \Rightarrow_{\sigma_1} H_1$ and $G_1 \Rightarrow_{\sigma_2} H_2$ is parallel independent (written $\mathsf{parindep}(\sigma_1, \sigma_2)$), if there are morphisms d_{12}, d_{21} such that $f_2 \circ d_{12} = m_1$, $f_1 \circ d_{21} = m_2$, $g_1 \circ d_{21} \models \phi_2$, and $g_2 \circ d_{12} \models \phi_1$.



Also, the GT steps in a set S are parallel independent, written $\mathsf{parindep}(S)$, when all pairs of GT steps in S are pairwise parallel independent.

A pair of GT steps $G_1 \Rightarrow_{\sigma_1} G_2$ and $G_2 \Rightarrow_{\sigma_2} G_3$ is sequentially independent (written $\mathsf{seqindep}(\sigma_1, \sigma_2)$), if there are morphisms d_{12}, d_{21} such that $f_2 \circ d_{12} = n_1, g_1 \circ d_{21} = m_2$, $f_1 \circ d_{21} \models \phi_2$, and $g_2 \circ d_{12} \models \mathsf{R}(\rho_1, \phi_1)$. [1]



In our k-induction procedure, we apply the operation linearize to a set of steps δ. In this case, some permutation of parallel independent steps in δ may not allow for a sequence of sequentially independent steps.

In particular, already in the first iteration, parallel independence of steps is not transitive (e.g. the pairs of steps $((s_1, a, s_0), (s_2, b, s_0))$ and $((s_2, b, s_0), (s_3, c, s_0))$ may be parallel independent in Figure 5.1b without the pair of steps $((s_1, a, s_0), (s_3, c, s_0))$ also being parallel independent). For example, given the rules (in simplified notation) $\rho_1 : \emptyset \Rightarrow a$, $\rho_2 : \emptyset \Rightarrow b$, $\rho_3 : a \Rightarrow ac$, and the GT steps $bc \Rightarrow_{\rho_1} abc$, $ac \Rightarrow_{\rho_2} abc$, and $ab \Rightarrow_{\rho_3} abc$, there is no step using ρ_3 with target graph bc showing that the steps using ρ_1 and ρ_3 are parallel dependent.

Moreover, parallel independence of steps is not necessarily preserved under the square construction (e.g. all pairs of distinct steps with target s_0 may be parallel independent in Figure 5.1b but some pair of distinct steps with common target may be parallel dependent in the second iteration). While parallel independence is preserved under square construction in this sense for graph transformation without ACs (see Lemma 2 below), this is not the case for graph transformation with ACs (see Example 1|p.37).

Lemma 2 (Preservation of Parallel Independence Without ACs)**.** For the case of GT steps using rules without ACs, we state that if the GT steps in $\{\sigma_1, \sigma_2, \sigma_3\}$ are pairwise parallel independent and the operation linearize constructs for $\{\sigma_1, \sigma_2\}$ the GT steps $\{\sigma_1, \sigma_{2b}\}$ and for $\{\sigma_1, \sigma_3\}$ the GT steps $\{\sigma_1, \sigma_{3b}\}$, then the GT steps in $\{\sigma_{2b}, \sigma_{3b}\}$ are parallel independent.

Proof. The proof of the Local Church-Rosser theorem (see [10, Theorem 4.7] for the case with ACs and [8, Theorem 5.12] for the case without ACs) provides a construction procedure for the operation linearize. The present lemma merely needs to show that the two applications of that procedure allow afterwards for the construction of the required morphisms to show that the steps σ_{2b} and σ_{3b} are parallel independent. First, we briefly recall *how* these two steps are obtained using the existing construction procedure without repeating the explanations for correctness. Afterwards, we show the existence of the required morphisms.

We begin with the parallel independence diagram from Definition 6 satisfying $f_2 \circ d_{12} = m_1$, $f_1 \circ d_{21} = m_2$, $g_1 \circ d_{21} \models \sigma_2$.rule.ac, and $g_2 \circ d_{12} \models \sigma_1$.rule.ac where

[1] Here, $\mathsf{R}((\ell, r, \phi), \phi) = \mathsf{L}((r, \ell, \top), \phi)$. Note that the AC of the rule is irrelevant for L and that our rules use only left-hand side ACs to ease the presentation overall.

the notation σ.rule.ac is used to extract the AC of the rule used in the step σ. However, as stated in the lemma, we consider here only rules without ACs, i.e., we require all ACs to be $\top$ meaning that they are trivially satisfied.

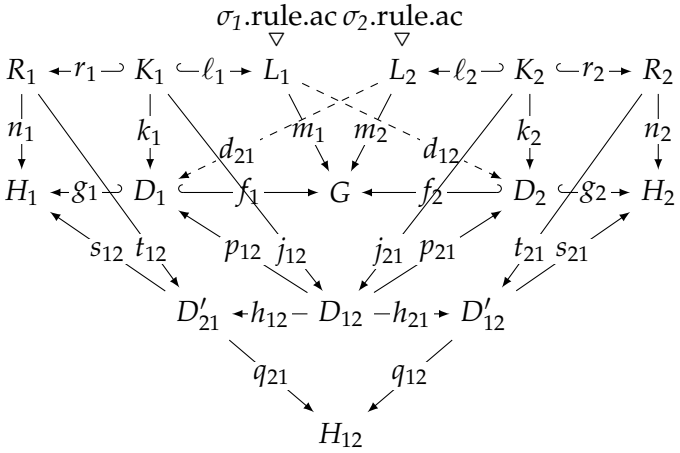
We follow [8, Theorem 5.12] for the case without ACs. We construct the pullback (D_{12}, p_{12}, p_{21}) of (f_1, f_2).

- The morphism j_{12} is obtained from the universal property of that pullback for $(k_1, d_{12} \circ \ell_1)$ satisfying $p_{12} \circ j_{12} = k_1$ and $p_{21} \circ j_{12} = d_{12} \circ \ell_1$.
- (Analogously to the previous item) The morphism j_2 is obtained from the universal property of that pullback for $(k_2, d_{21} \circ \ell_2)$ satisfying $p_{21} \circ j_{21} = k_2$ and $p_{12} \circ j_{21} = d_{21} \circ \ell_2$.

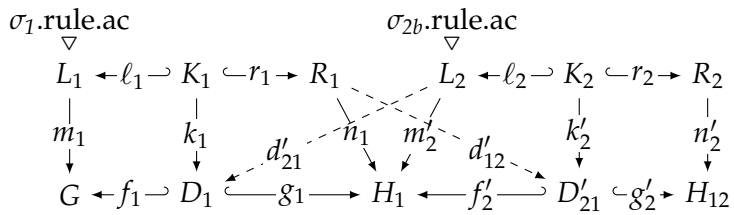
Then, (D_2, d_{12}, p_{21}) is a pushout of the pair (ℓ_1, j_{12}), (D_1, d_{21}, p_{12}) is a pushout of the pair (ℓ_2, j_{21}), and (G, f_1, f_2) is a pushout of the pair (p_{12}, p_{21}).

- We construct the pushout $(D'_{21}, h_{12}, t_{12})$ of (j_{12}, r_1). The morphism s_{12} is obtained from the universal property of that pushout for (n_1, g_1) satisfying $s_{12} \circ t_{12} = n_1$ and $s_{12} \circ h_{12} = g_1 \circ p_{12}$. Then, (H_1, s_{12}, g_1) is a pushout of (h_{12}, p_{12}).
- (Analogously to the previous item) We construct the pushout $(D'_{12}, h_{21}, t_{21})$ of (j_{21}, r_2). The morphism s_{21} is obtained from the universal property of that pushout for (n_2, g_2) satisfying $s_{21} \circ t_{21} = n_2$ and $s_{21} \circ h_{21} = g_2 \circ p_{21}$. Then, (H_2, s_{21}, g_2) is a pushout of (h_{21}, p_{21}).

We construct the pushout (H_{12}, q_{12}, q_{21}) of (h_{12}, h_{21}).



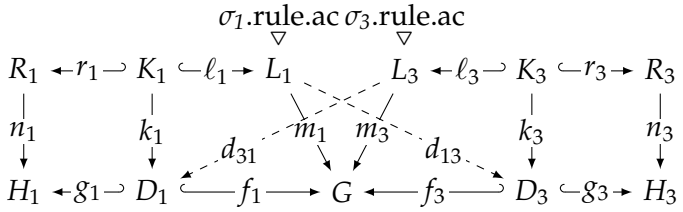
We obtain the two desired sequentially independent steps σ_1 and σ_{2b} (where σ_{2b} and σ_2 use the same rule) as follows (we omit the second ordering as it is not relevant here). For the sequential independence diagram below, we use $m'_2 = g_1 \circ d_{21}$, $k'_2 = h_{12} \circ j_{21}$, $n'_2 = q_{12} \circ t_{21}$, $f'_2 = s_{12}$, and $g'_2 = q_{21}$.



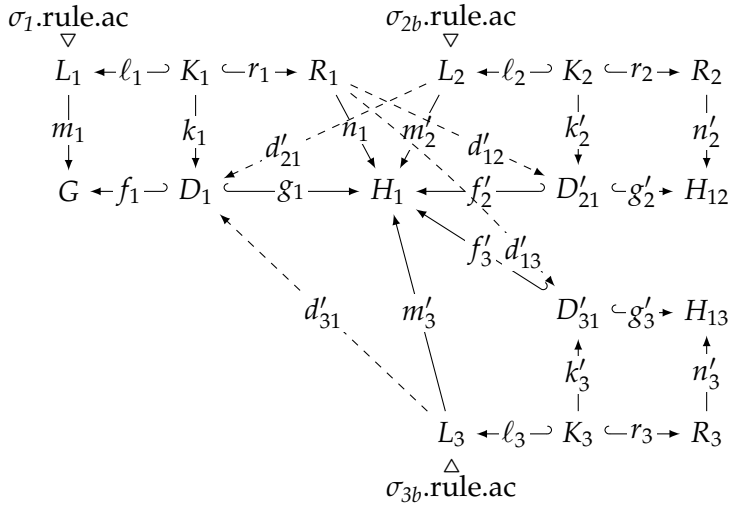
The two steps are sequentially independent using $d'_{12} = t_{12}$ (satisfying $f'_2 \circ d'_{12} = n_1$ since $s_{12} \circ t_{12} = n_1$ as derived above) and $d'_{21} = d_{21}$ (satisfying $g_1 \circ d'_{21} = m'_2$ by unfolding).

For the extended case with ACs as in [10, Theorem 4.7], we note that it is proven there that $f_1 \circ d'_{21} \models \sigma_{2b}$.rule.ac and $g'_2 \circ d'_{12} \models \mathsf{R}(\sigma_1.\text{rule}, \sigma_1.\text{rule.ac})$, showing that the two steps are sequentially independent also for the case with ACs. However, we do not use these two GCs as we assume all ACs to be $\top$.

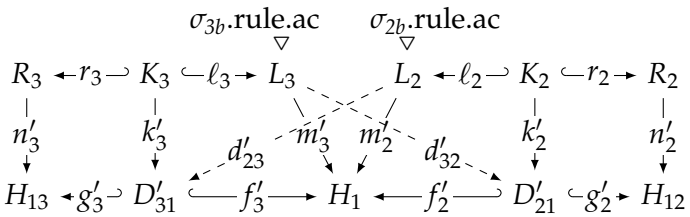
We now assume another step σ_3 that is parallel independent to the previously considered steps σ_1 and σ_2.



As for σ_1 and σ_2, we can now obtain σ_{3b} as in the following diagram extending the diagram from above for σ_1 and σ_{2b}.

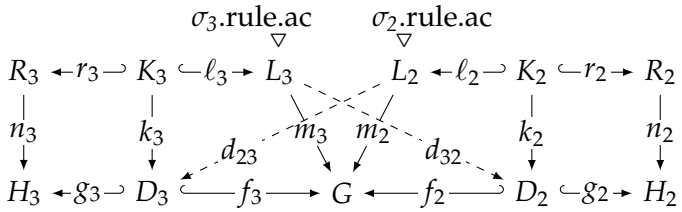


Removing σ_1 for clarity from this diagram gives us a parallel independence diagram for σ_{2b} and σ_{3b} where we need to show the existence of d'_{23}, d'_{32} satisfying $f'_2 \circ d'_{32} = m'_3$, $f'_3 \circ d'_{23} = m'_2$, $g'_3 \circ d'_{23} \models \sigma_{2b}$.rule.ac, and $g'_2 \circ d'_{32} \models \sigma_{3b}$.rule.ac. Again, we note that all ACs are assumed to be $\top$.



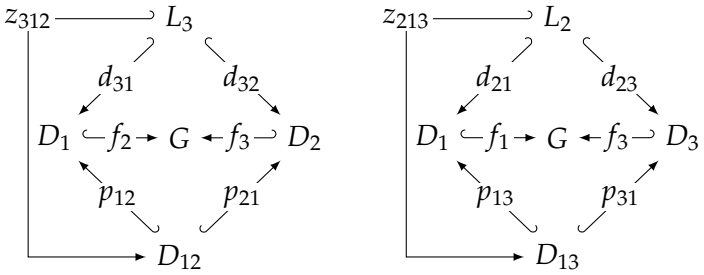
Note that we have used the parallel independence among σ_1 and σ_2 and the parallel independence among σ_1 and σ_3 but not yet the parallel independence among σ_2 and

σ_3 given in the following diagram satisfying $f_3 \circ d_{23} = m_2$, $f_2 \circ d_{32} = m_3$, $g_2 \circ d_{32} \models \sigma_3$.rule.ac, and $g_3 \circ d_{23} \models \sigma_2$.rule.ac. Again, all ACs are assumed to be $\top$; hence, we do not have to verify the last two GC satisfaction statements.



We now obtain two morphisms.

- The morphism $z_{312} : L_3 \rightarrowtail D_{12}$ is obtained from the universal property of the pullback (D_{12}, p_{12}, p_{21}) using (d_{31}, d_{32}) satisfying $p_{12} \circ z_{312} = d_{31}$ and $p_{21} \circ z_{312} = d_{32}$. The morphism $h_{12} \circ z_{312} : L_3 \rightarrowtail D'_{21}$ is now used for d'_{32}. See the left side of the picture below.
- (Analogously to the previous item) The morphism $z_{213} : L_2 \rightarrowtail D_{13}$ is obtained from the universal property of the pullback (D_{13}, p_{13}, p_{31}) using (d_{21}, d_{23}) satisfying $p_{13} \circ z_{213} = d_{21}$ and $p_{31} \circ z_{213} = d_{23}$. The morphism $h_{13} \circ z_{213} : L_2 \rightarrowtail D'_{31}$ is now used for d'_{23}. See the right side of the picture below.



It remains to be verified that $f'_2 \circ d'_{32} = m'_3$ and $f'_3 \circ d'_{23} = m'_2$.

- We first verify that $f'_3 \circ d'_{23} = m'_2$:

$$
\begin{aligned}
& f'_3 \circ d'_{23} = m'_2 \\
& \Longleftarrow (\text{using } d'_{23} = h_{13} \circ z_{213}) \\
& f'_3 \circ h_{13} \circ z_{213} = m'_2 \\
& \Longleftarrow (\text{using } m'_2 = g_1 \circ d'_{21})) \\
& f'_3 \circ h_{13} \circ z_{213} = g_1 \circ d'_{21} \\
& \Longleftarrow (\text{using } f'_3 = s_{13}) \\
& s_{13} \circ h_{13} \circ z_{213} = g_1 \circ d'_{21} \\
& \Longleftarrow (\text{using } s_{13} \circ h_{13} = g_1 \circ p_{13}) \\
& g_1 \circ p_{13} \circ z_{213} = g_1 \circ d'_{21} \\
& \Longleftarrow (\text{composing with } g_1) \\
& p_{13} \circ z_{213} = d'_{21}
\end{aligned}
$$

$$
\begin{aligned}
&\Longleftarrow (\text{using } d'_{21} = d_{21}) \\
&p_{13} \circ z_{213} = d_{21} \\
&\Longleftarrow (\text{using } p_{13} \circ z_{213} = d_{21}) \\
&\mathit{true}
\end{aligned}
$$

- (Analogously to the previous item) We now verify that $f'_2 \circ d'_{32} = m'_3$:

$$
\begin{aligned}
&f'_2 \circ d'_{32} = m'_3 \\
&\Longleftarrow (\text{using } d'_{32} = h_{12} \circ z_{312}) \\
&f'_2 \circ h_{12} \circ z_{312} = m'_3 \\
&\Longleftarrow (\text{using } m'_3 = g_1 \circ d'_{31})) \\
&f'_2 \circ h_{12} \circ z_{312} = g_1 \circ d'_{31} \\
&\Longleftarrow (\text{using } f'_2 = s_{12}) \\
&s_{12} \circ h_{12} \circ z_{312} = g_1 \circ d'_{31} \\
&\Longleftarrow (\text{using } s_{12} \circ h_{12} = g_1 \circ p_{12}) \\
&g_1 \circ p_{12} \circ z_{312} = g_1 \circ d'_{31} \\
&\Longleftarrow (\text{composing with } g_1) \\
&p_{12} \circ z_{312} = d'_{31} \\
&\Longleftarrow (\text{using } d'_{31} = d_{31}) \\
&p_{12} \circ z_{312} = d_{31} \\
&\Longleftarrow (\text{using } p_{12} \circ z_{312} = d_{31}) \\
&\mathit{true}
\end{aligned}
$$

□

The following example shows that the statement from Lemma 2 does not hold for the case of rules with ACs.

Example 1 (Preservation of Parallel Independence With ACs)**.** The use of arbitrary negative application conditions suffices to show that the statement given in Lemma 2 does not hold for the case of rules with ACs. For example, if there are three rules ρ_1, ρ_2, and ρ_3 where $L_i = K_i = \emptyset$ $(1 \leq i \leq 3)$, the AC of ρ_1 states the non-existence of two nodes $b : B$ and $c : C$, the AC of ρ_2 states the non-existence of two nodes $a : A$ and $c : C$, the AC of ρ_3 states the non-existence of two nodes $a : A$ and $b : B$, and ρ_1, ρ_2, and ρ_3 add a single node $a : A$, $b : B$, and $c : C$, respectively. For the graph $G = \emptyset$, all rules are applicable for a unique match $(m : \emptyset \rightarrowtail \emptyset)$. Also, each pair of these GT steps is parallel independent. However, when moving the GT steps for ρ_2 and ρ_3 over the GT step for rule ρ_1, we note that the two resulting GT steps using the two rules ρ_2 and ρ_3 are not parallel independent.

Current Technical Reports of the Hasso-Plattner-Institute

Vol.	ISBN	Title	Authors/Editors
142	978-3-86956-524-8	**Quantum computing from a software developers perspective**	Marcel Garus, Rohan Sawahn, Jonas Wanke, Clemens Tiedt, Clara Granzow, Tim Kuffner, Jannis Rosenbaum, Linus Hagemann, Tom Wollnik, Lorenz Woth, Felix Auringer, Tobias Kantusch, Felix Roth, Konrad Hanff, Niklas Schilli, Leonard Seibold, Marc Fabian Lindner, Selina Raschack
141	978-3-86956-521-7	**Tool support for collaborative creation of interactive storytelling media**	Paula Klinke, Silvan Verhoeven, Felix Roth, Linus Hagemann, Tarik Alnawa, Jens Lincke, Patrick Rein, Robert Hirschfeld
140	978-3-86956-517-0	**Probabilistic metric temporal graph logic**	Sven Schneider, Maria Maximova, Holger Giese
139	978-3-86956-514-9	**Deep learning for computer vision in the art domain : proceedings of the master seminar on practical introduction to deep learning for computer vision, HPI WS 20/21**	Christian Bartz, Ralf Krestel
138	978-3-86956-513-2	**Proceedings of the HPI research school on service-oriented systems engineering 2020 Fall Retreat**	Christoph Meinel, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Erwin Böttinger, Christoph Lippert, Christian Dörr, Anja Lehmann, Bernhard Renard, Tilmann Rabl, Falk Uebernickel, Bert Arnrich, Katharina Hölzle
137	978-3-86956-505-7	**Language and tool support for 3D crochet patterns virtual crochet with a graph structure**	Klara Seitz, Jens Lincke, Patrick Rein, Robert Hirschfeld
136	978-3-86956-504-0	**An individual-centered approach to visualize people's opinions and demographic information**	Wanda Baltzer, Theresa Hradilak, Lara Pfennigschmidt, Luc Maurice Prestin, Moritz Spranger, Simon Stadlinger, Leo Wendt, Jens Lincke, Patrick Rein, Luke Church, Robert Hirschfeld

ISBN 978-3-86956-531-6
ISSN 1613-5652