EVALUATING TEMPORAL QUERIES OVER HISTORY-AWARE ARCHITECTURAL RUNTIME MODELS

DISSERTATION

von

LUCAS SAKIZLOGLOU, M. SC.

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)

in der Wissenschaftsdisziplin
Praktische Informatik

eingereicht an der
Digital-Engineering-Fakultät
des Hasso-Plattner-Instituts und
der Universität Potsdam

Tag der Verteidigung: 21. Juli 2023

Potsdam, 2022

Lucas Sakizloglou: Evaluating Temporal Queries over History-aware Architectural Runtime Models, 2022

Advisor:
Prof. Dr. Holger Giese
Hasso Plattner Institute, University of Potsdam, Germany

Reviewers:
Dr. Nelly Bencomo, Associate Professor
University of Durham, United Kingdom

Prof., PhD, habil. DSc Dániel Varró
Linköping University, Sweden

The author was partially funded by the German Research Foundation (DFG) under the project GRK 1651: Service-oriented Architectures for the Integration of Software-based Processes, Exemplified by Health Care Systems and Medical Technology (SOAMED).

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-60439
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-604396

ABSTRACT

In model-driven engineering, the adaptation of large software systems with dynamic structure is enabled by architectural runtime models. Such a model represents an abstract state of the system as a graph of interacting components. Every relevant change in the system is mirrored in the model, and triggers an evaluation of model queries which search the model for structural patterns that should be adapted. This thesis focuses on a type of runtime models where the expressiveness of the model and model queries is extended to capture past changes and their timing. These history-aware models and temporal queries enable more informed decision-making during adaptation, as they support the formulation of requirements on the evolution of the pattern that should be adapted. However, evaluating temporal queries during adaptation poses significant challenges. First, it implies the capability to specify and evaluate requirements on the structure as well as the ordering and timing in which structural changes occur. Then, query answers have to reflect that the history-aware model represents the architecture of a system whose execution may be ongoing, and thus answers may depend on future changes. Finally, query evaluation needs to be adequately fast and memory-efficient despite the increasing size of the history—especially for models that are altered by numerous, rapid changes.

The thesis presents a query language and a querying approach for the specification and evaluation of temporal queries. These contributions aim to cope with the challenges of evaluating temporal queries at runtime, a prerequisite for history-aware architectural monitoring and adaptation which has not been systematically treated by prior model-based solutions. The distinguishing features of our contributions are: the specification of queries based on a temporal logic which encodes structural patterns as graphs; the provision of formally precise query answers which account for timing constraints and ongoing executions; the incremental evaluation which avoids the re-computation of query answers after each change; and the option to discard history that is no longer relevant to queries. The query evaluation searches the model for occurrences of a pattern whose evolution satisfies a temporal logic formula. Therefore, besides model-driven engineering, another related research community is runtime verification. The approach differs from prior logic-based runtime verification solutions by supporting the representation and querying of structure via graphs and graph queries respectively, which is more efficient for queries with complex patterns. We present a prototypical implementation of the approach and measure its speed and memory consumption in monitoring and adaptation scenarios from two application domains, with executions of an increasing size. We assess scalability by a comparison to the state-of-the-art from both related research communities. The implementation yields promising results, which pave the way for sophisticated history-aware self-adaptation solutions and indicate that the approach constitutes a highly effective technique for runtime monitoring on an architectural level.

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

Advanced software systems such as IoT comprise numerous interconnected, heterogeneous, and autonomous entities, rendering their runtime behavior unpredictable. Consequently, ensuring that these systems continuously fulfill their requirements calls for sophisticated *adaptation schemes* which can monitor the actual runtime behavior and, if necessary, adapt it. A *Runtime Model* (RTM) [28] is an artifact that stems from *Model-driven Engineering* (MDE). The RTM captures a representation of the state of a running software system at a desired level of abstraction and is *causally connected* to the system, i.e., any relevant change to the system is reflected in the model and vice versa; the model can, therefore, support monitoring and adaptation at runtime.

For adaptation of a complex system with dynamic behavior, the appropriate level of abstraction of the model is generally considered to be the software architecture [see 18, 153], as it provides a global perspective on the system and captures the most important system-level behaviors and properties [62]. Such an *architectural* RTM represents the system state as a snapshot of the architecture consisting of *components*, i.e., computational elements, and *connectors*, i.e., interactions between components [120]. Monitoring, i.e., searching the model for situations of interest which should be addressed via adaptation, is performed by evaluating a *(model) query* over the snapshot: Every relevant change in the system is mirrored in the model and triggers an evaluation of the query; the query searches the model for occurrences of a structural pattern describing an issue; a query answer contains matches, i.e., found occurrences of the pattern. Adaptation encompasses monitoring, and may use a match as a precondition that triggers an *adaptation action*, i.e., the addition, removal, or replacement of components and connectors [112] in the match, which aims to resolve the issue. The adaptation action is performed by an in-place transformation of the model [67] which, by virtue of the causal connection, is mirrored in the system.

This thesis focuses on a type of architectural RTMs where the expressiveness of the model and model queries is extended to capture *history*, i.e., past changes to the architecture and their timing. *History-aware models* and *temporal queries* enable more informed decision-making during adaptation as they allow for the expression of expectations on the evolution of a pattern over time. Therefore, history-aware models and temporal queries can be used for checking assertions about the past or future, detecting recurrent behaviors, or predicting future trends [20]. Moreover, history-aware models can be employed by advanced systems for certain application domains where consideration of temporal data when examining whether an expectation is fulfilled is mandatory, e.g., in healthcare, where timing constraints are always present in medical guidelines and their fulfillment is key to the successful completion of medical procedures [37].

1.2 PROBLEM STATEMENT

The inherent support for history, as defined above, has been identified as one of the grand challenges for modern MDE solutions [31]. For architectural RTMs in particular, representing history requires a compact encoding of past snapshots over which changes to model can be tracked [80]; this encoding should capture the lifespans of entities, i.e., the time intervals for which they exist in the model [35]. The query language has to support constraints on the ordering as well as the real time in which structural changes occur and the inclusion of timing information in query answers [27]. The use of model queries for monitoring requires formally precise semantics [33] such that the soundness of answers can be ensured; for queries with future timing constraints, the semantics have to reflect that that queries may be evaluated over an evolving RTM, i.e., an unfinished system execution, and thus the satisfaction of these constraints may depend on future changes to the model. Finally, as the history accumulates over time, the evaluation of temporal queries should remain scalable [61] by using incremental evaluation techniques and measures for discarding history which is irrelevant to queries, thereby reducing the memory consumption [10, 27]. This last challenge is exacerbated by the fact that, typically, an RTM is queried at runtime and monitoring and adaptation solutions depend on the query answer for taking a remedial action or planning an adaptation.

Based on these challenges, we elicit the following requirements for solutions which support the representation and querying of a history-aware RTM for monitoring and adaptation: *R1*—a *history encoding* which compactly captures multiple snapshots, and enables tracing the timing of changes in the lifespans of entities; *R2—a temporal query language* which supports (i) the specification of queries on the structure as well as the ordering and timing in which structural changes occur (ii) the provision of answers with timing information; *R3—sound answers* based on formally precise semantics, even if the query has future timing constraints and the evaluation is performed over an unfinished execution; *R4*—an *incremental evaluation* technique which enables fast query evaluation; *R5*—a *memory-efficient encoding* which may reduce the memory consumption and expedite query evaluations.

Despite their potential to enable more informed decision-making, and perhaps because of the challenges involved, monitoring and adaptation based on the evaluation of temporal queries over history-aware RTMs have been generally neglected by the otherwise extensive research on RTMs [18]. Only recently did solutions emerge that specifically support the storage and querying of the history of an architectural RTM. The state-of-the-art [61] does not have formally precise semantics and focuses on offline use-cases, e.g., postmortem analysis, therefore lacking an incremental evaluation technique which would allow for scalable query evaluation of temporal queries at runtime.

The history of an architectural RTM may be also encoded and queried by other model-based technologies. State-of-the-art solutions for non-history-aware architectural RTMs, e.g., [63, 155], support the incremental evaluation of purely structural queries, and are shown to be scalable for large systems with dynamic architectures; although history and temporal queries could be indirectly supported by these solutions, i.e., via a manual encoding of timing information, this task is not trivial and the encoding would remain ad hoc.

Model-versioning solutions, e.g., [76, 80], could also encode the history of an RTM. There is, however, a considerable difference between the objectives of model versioning and those of RTM-based monitoring, which renders those solutions similarly unsuitable for the repetitive evaluation of temporal queries at runtime. For example, these solutions often assume that queries will mostly concern a single timestamp, i.e., a specific version, and they are therefore optimized for such queries and are less suitable for the evaluation of pattern-based queries that refer to a period of time [76].

Seen in a broader context, the task of monitoring whether a given expectation with temporal constraints is fulfilled by a possibly unfinished execution resembles the focus of the research community known as *Runtime Verification* (RV) [11]. RV represents the system behavior by sequences of events at some level of abstraction. An online algorithm is then used to verify whether (some prefix of) the sequence satisfies a given property [13], i.e., a formal statement. The most advanced RV solutions combine an incremental monitoring algorithm with behavior representations which contain only information that is relevant to the monitored property. Therefore, they are adequately fast and memory-efficient, even in the face of numerous, fast-paced events.

However, although similarities between RTM-based monitoring and RV can be observed, so can fundamental differences. An RTM is a causally-connected snapshot of the architecture, which typically is the product of a broader MDE context. Query answers over the RTM are supposed to be further utilized at once within that context—for instance, for self-adaptation. Contrary to an RTM, representations of the system state in RV solutions are created ad hoc and are typically inaccessible by other tools or end-users [77], which may render RV solutions impractical for use within an MDE context, e.g., it may hinder synergy with other model-based technologies. Moreover, as architectural models are typically not their primary focus, RV solutions do not match the efficiency of relevant model-based solutions in representing and querying structures [33, 46]. These leads indicate that RV solutions cannot replace model-based solutions in architectural adaptation schemes.

In conclusion, both in MDE and RV, there is a lack of a systematic treatment of temporal query evaluation over history-aware architectural RTM, which fully supports requirements considered by the relevant literature to be necessary for architectural monitoring and adaptation.

1.3 OBJECTIVES AND CONTRIBUTIONS

Our contributions aim at presenting such a systematic treatment, which fulfills the requirements mentioned previously and thus accomplishes the following two objectives: An appropriate *formalization* of the problem of incrementally evaluating a temporal model query over a history-aware architectural RTM, thereby increasing confidence in the soundness of answers; a solution which affords increased *scalability* compared to the state-of-the-art, thereby enabling sophisticated history-aware adaptation and a highly effective approach to runtime monitoring on an architectural level. Scalability denotes the capability of a solution to mitigate the dependence of query evaluation times and memory consumption on the size of the history.

Following is a list of our contributions and their distinguishing features.

C1 An encoding of history-aware architectural RTMs, called *Runtime Model with History* (RTM^H). Following the common practice of representing an RTM as a graph, the RTM^H is a graph which corresponds to a sequence of timestamped graphs capturing previous snapshots of the RTM. Each entity in the RTM^H has a creation and deletion timestamp which capture its lifespan. This contribution fulfills the requirement for a compact encoding of the model history (*R1*).

C2 A query language for the specification of temporal queries over an RTM^H (C2.1). The language is based on an interval-based temporal logic with inherent support for graphs and, therefore, supports statements on the model structure as well as the ordering and (quantitative) timing in which structural changes occur. Queries are characterized by a pattern and a formula defined in the logic. The answer to a query contains pattern matches in the RTM^H whose evolution satisfies the formula, paired with their *temporal validity*, i.e., their lifespan intersected with the period for which they satisfy the formula (C2.2). Contributions C2.1 and C2.2 fulfill the requirement for an adequately expressive query language with intuitive answers (*R2*).

Based on a novel three-valued interpretation of the logic (C2.3), the language supports two types of answers which respectively assume that the evaluation of a query is performed over an RTM^H corresponding to a finished and an unfinished execution (C2.4). We provide proofs on the soundness of the computation of the temporal validity by both types of answers. The contributions C2.3 and C2.4 emphasize the fulfillment of the requirement for sound answers (*R3*), to which all our formal results contribute.

C3 A querying approach, named InTempo, which decomposes a temporal query into a directed acyclic graph of simple sub-queries based on a novel operationalization framework (C3.1); edges correspond to dependencies between sub-queries. Sub-queries store their answers in-between query evaluations and are reevaluated only if a change to the RTM^H concerns them, thus enabling incremental evaluation. We show that the decomposition produces a result that is equivalent to the query answer.

InTempo offers the option of deriving a window during which deleted entities in the RTM^H are relevant to query evaluations; the window is derived based on the timing constraints of the formula of the query; InTempo then prunes entities outside the window thus reducing the size of the RTM^H (C3.2). We show that over a given execution, i.e., a sequence of RTM^H instances, evaluations over the pruned and complete versions return the same answers, thus pruning does not affect *R3*. The contributions C3.1 and C3.2 fulfill the requirements for incremental query evaluation (*R4*) and a memory-efficient encoding (*R5*).

C4 A prototypical implementation of InTempo as a plugin of the well-known Eclipse Modeling Framework [51, 149]. The plugin supports syntax validation of queries specified in the presented query language and the modification of an RTM^H based on an event log containing system changes.

1.4 PLAN FOR FULFILLMENT AND ASSESSMENT OF OBJECTIVES

Regarding the objective of an appropriate formalization, contrary to the state-of-the-art in history-aware model-based technologies, we provide proofs on the soundness of an answer to a temporal query as well as the method of the answer computation. Our contributions are formally underpinned by the well-established graph transformation theory [53] and related formal results: the Metric Temporal Graph Logic [68, 143], which constitutes joint work of the author; and a formal operationalization framework for structural queries based on graph transformation [23], which we extend to support temporal queries in the introduced query language.

Regarding the objective of increased scalability, we conduct an experimental evaluation based on an implementation of InTempo and two experiments stemming from different application domains. In both experiments, we measure the query evaluation times and memory consumption of InTempo over histories of an increasing size. The performance of InTempo is compared to the state-of-the-art of RV and MDE. In the first experiment, the implementation of InTempo is integrated with a feedback control loop which instruments self-adaptation, i.e., a self-adaptation engine [93], and evaluated via simulations based on a case-study of a smart medical system, a real medical guideline [126], and a combination of real and synthetic event logs. The logs are used for simulations in which query answers from InTempo are used for performing straightforward adaptations. In the second experiment, we use data generated by the Social Network Benchmark by the Linked Data Benchmark Council [103] to test the performance of the implementation with more complex queries over larger and more complex graph structures.

1.5 OUTLINE

Following is an outline of the contents of the thesis.

Chapter 2 presents the foundations of the contributions, namely architectural RTMs, the graph representation used for RTMs, graph queries, and graph transformation. This chapter also introduces the temporal graph logic that serves as a basis for the query language.

Chapter 3 introduces the RTM^{H} (contribution C1, Section 3.1) in technical detail. Moreover, this chapter introduces the query language, i.e., the contribution C2.1. Section 3.2 presents the syntax and the basic concepts, including the temporal validity (contribution C2.2, Section 3.2). Section 3.3 presents the novel three-valued interpretation of the temporal logic (C2.3) which form the basis for the query language semantics over a finished and an unfinished execution (C2.4). The chapter concludes with a summary of the introduced concepts in Section 3.4.

Chapter 4 introduces InTempo. The querying approach comprises the following three inter-dependent operations: the *Operationalization*, presented in Section 4.1, operationalizes a temporal query in the presented language by decomposing it into a network of simple sub-queries (C3.1); the (query) *Evaluation*, presented in Section 4.2 executes the network, thus performing the actual query evaluation; the (optional) *Maintenance*, presented in Section 4.3, derives a time window based on which the RTM^H is pruned, and returns a projected answer set over the pruned RTM^H (C3.2). Section 4.4 demonstrates the integration of the querying approach into a self-adaptation engine, thereby rendering the engine capable of instrumenting history-aware self-adaptation. The chapter concludes with a summary of the introduced concepts in Section 4.5.

Chapter 5 presents the prototypical implementation of InTempo in Section 5.1 (C4). Moreover, this chapter contains the experimental evaluation of the implementation for the two case-studies (see Section 5.2 and Section 5.3), and the comparison of the results to the performance of the state-of-the-art—see Section 5.4. The chapter concludes in Section 5.5 with an assessment of the fulfillment of the objective for increased scalability, a reflection on advantages and limitations of InTempo compared to the state-of-the-art, and a discussion of the threats to validity of the experimental evaluation as well as the measures taken for their mitigation.

Chapter 6 discusses related work and Chapter 7 concludes the thesis and discusses plans for future work.

Appendix A is a technical supplement to the thesis, containing technical introductions to important concepts and proofs. Appendix B provides additional information on the experimental evaluation. Finally, Appendix C contains a list of published articles and papers in peer-reviewed journals and proceedings, respectively, which were used as the basis for the thesis.

2 FOUNDATIONS

This chapter presents the foundations of our contributions.

Section 2.1 briefly iterates the fundamentals behind RTMs, introduces the specific structural architectural RTMs used in the thesis, presents the running example, and defines a history representation based on a sequence of RTMs.

As it represents structure, an architectural RTM is amenable to a graph-based encoding. We follow the common practice of representing an RTM as a typed attributed graph. Section 2.2 presents all the formal graph-related concepts employed in our contributions, i.e., typed attributed graphs, graph queries, graph transformation, and the operationalization of graph queries based on graph transformation and incremental techniques.

Section 2.3 presents the *Metric Temporal Graph Logic* (MTGL), a metric temporal logic with direct support for graphs; MTGL enables the formulation of quantitative temporal requirements on the evolution of graph structures and constitutes as such the basis of the temporal model query language presented by the thesis.

2.1 RUNTIME MODELS

2.1.1 *Background*

In software engineering, a *model* is a representation of an *original*, i.e., a system or aspect thereof; the model is built for a specific purpose, e.g., accomplishing a task or simplifying a complex problem, and captures only those characteristics of the original that are relevant to its purpose [64]. Familiar examples of models are an architectural model, which captures the system structure in terms of components and their interactions [120], and a finite-state machine, which captures possible behaviors of a system.

Model-driven Engineering (MDE) is a software engineering methodology in which, during the development life-cycle, i.e., the design-time, development tasks are either automated or performed by systematically transforming unambiguous *development models* into software artifacts [see 30]. For instance, models may be used for code generation, automated testing, and refactoring, as well as static analysis, i.e., the verification—based on a model—that certain behaviors are never observed. Unambiguity refers to the models being defined according to a concrete syntax.

A *Runtime Model* (RTM) builds on information available in development models but instead focuses on tasks and problems related to the *runtime* of a system, i.e., its execution. RTMs act as an interface to *monitor* and *manage* the system during its execution, while concealing the complexity these tasks may entail [59]. For instance, RTMs may be used to support monitoring of the state of the system, ascertaining whether the system meets its requirements, or adapting its behavior during runtime. To these ends, it is typically required that RTMs are *self-representations*, i.e., modeling a system which can observe changes to itself, and *causally-connected*, i.e., a change in the system is mirrored in the RTM and vice-versa [28].

These characteristics are founded *on computational reflection* [104] and ensure that an RTM reflects the latest state of the associated system—which enables monitoring—and that adaptations on the system can be performed on the model and are then mirrored in the system—which enables management. In practice, the causal connection may be perceived as a layer that links the model space to the runtime space [117], that is, an infrastructure that is equipped to translate a change to the RTM into a change to the system and vice-versa.

As an example, consider an RTM which provides a view on the current architecture of a system, i.e., activated components and connectors. Connectors represent dependencies or currently established network connections. Upon a change to the system structure, i.e., a deactivated component, the RTM is updated to reflect said change. Conversely, a desired modification to the structure, i.e., an activation of a new component, can be performed on the RTM which, via the causal connection, is then mirrored in the associated system.

2.1.2 *Runtime Models in this Thesis*

The recent survey on research based on RTMs in [18] paints the picture of a plethoric yet fragmented landscape where several *types* of models are employed to model a diverse array of *artifacts* for a variety of *purposes*: For instance, structural RTMs (the type) are employed to model the system architecture (the artifact) for the purpose of adapting the architecture at runtime (the purpose); goal RTMs (the type) are employed to model system requirements (the artifact) for the purpose of assuring selected non-functional properties of the associated system (the purpose). We position the RTMs in this thesis within this landscape based on the same dimensions, i.e., type, artifact, and purpose.

Regarding the type, this thesis focuses on structural RTMs which are essentially component graphs consisting of components, connectors, and information about components. Regarding the artifact, we employ structural RTMs to capture a *snapshot*, i.e., the current state, of the architecture of a system, a practice which is commonplace, e.g., [56, 63, 117, 157]. In a snapshot, components correspond to processing elements of the system, and connectors to communication or interdependencies among components [120]. Henceforth, we refer to RTMs simply as architectural. As usual, the RTM is assumed to represent a soft real-time system [see 34] and, moreover, the modeled architecture is extended by information on the *context* of the system, i.e., everything in the operating environment that affects the properties of the system and its behavior [138], which facilitates analysis [158]. For the creation of unambiguous RTMs we employ a *metamodel*, i.e., a syntax which defines valid RTM instances [24].

Regarding their purpose, RTMs serve primarily as a *knowledge base* for adaptation schemes. For example, the well-known MAPE-K feedback loop [93] maintains an RTM as part of its knowledge on the current state of the system architecture; the loop uses the RTM to verify whether the system exhibits a desired behavior, and, if necessary, to adapt the system—for which the architecture is considered to be the appropriate level of abstraction [62, 120].

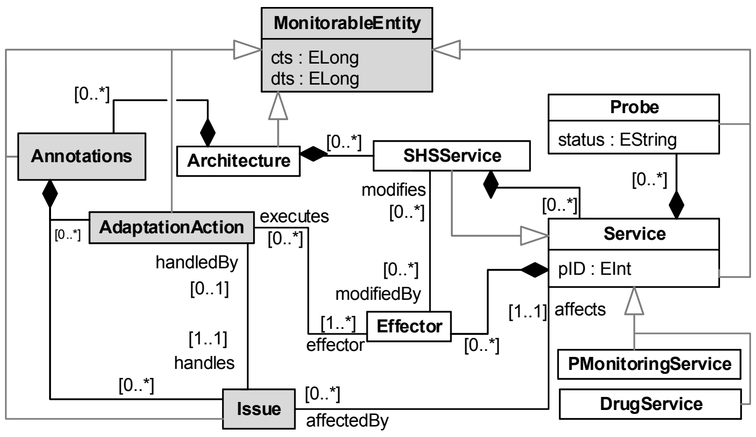


Figure 2.1: Metamodel of the SHS (excerpt)

The adaptation is accomplished by well-defined *rules*. The rules consist of a pre-condition which describes a condition over a part of the architecture, i.e., a substructure, which should be adapted, and a post-condition which describes architectural changes, e.g., addition or removal of components, in case occurrences of the substructure are found. The architectural changes are performed by *inline model transformation* [67], i.e., by transforming the RTM in place. Changes induced by the transformation are assumed to be mirrored in the system via causal connection.

Example 2.1.1 (smart hospital system – running example)**.** *Examples as well as part of the experimental evaluation in the thesis are based on a case-study of a service-based* Smart Healthcare System *(SHS). The SHS is based on smart medical environments [130] where sensors periodically collect physiological measurements from patients, i.e., data such as temperature, heart-beat, and blood pressure. Depending on the collected patient measurements, certain medical procedures are automated and performed by devices, such as smart pumps administering medicine—as otherwise a clinician would be doing. Figure 2.1 depicts the meta-model of the SHS based on the* Ecore *metamodeling syntax [51]. The metamodel draws from the well-known class diagram, therefore a component is modeled as a* class *and a component connector as a* reference*. The SHS metamodel is based on the exemplar of a service-based medical system in [161], and captures the architecture of a running SHS as an instance of the* `Architecture` *class.*

In the SHS, services are invoked by a main service called `SHSService` *to collect measurements from patient sensors, i.e.,* `PMonitoringService`*, or take medical actions via smart medical devices such as a pump, i.e.,* `DrugService`*. Instances of the* `SHSService` *correspond to a smart medical environment, i.e., a room in a hospital, enhanced with sensors and smart devices as well as the appropriate infrastructure. Invocations of services are triggered by effectors (*`Effector`*) and invocation results are tracked via monitoring probes (*`Probe`*) that are attached on* `Service`*s.* `Probe`*s are generated periodically or upon events in the real world. Each* `Probe` *has a* status *attribute whose value depends on the type of* `Service`*. Each* `Service` *has a* pID *attribute which identifies the patient for whom the* `Service` *is invoked. Entities in gray are explained in later chapters.*

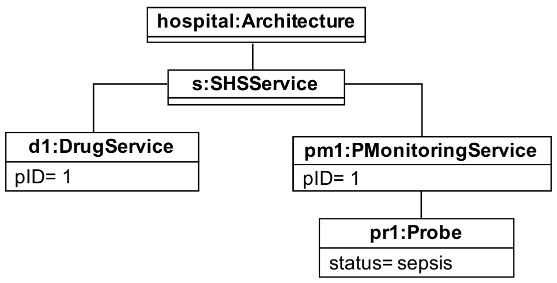


Figure 2.2: A rudimentary RTM based on the metamodel in Figure 2.1

See Figure 2.2 for an example of a rudimentary RTM instance (capturing a fragment of the `Architecture` *instance) based on the metamodel in Figure 2.1. The RTM shows that: an instance of the SHS has been created (entity* `hospital`*) and an instance of the main service has been invoked (entity* `s`*), i.e., a smart room has been activated; an instance of the monitoring service has been invoked (entity* `pm1`*), i.e., a sensor has been attached to a patient in the room; a probe has been generated by the monitoring service (entity* `pr1`*), i.e., a reading has been performed by the sensor; an instance of a drug service has been invoked (entity* `d1`*), i.e., a pump which can administer drugs has been attached to a patient in the room.*

2.1.3 *History Representation Based on Runtime Models*

We identify the system behavior with a (possibly infinite) sequence of instantaneous *events* which represent observable actions or state changes made by the system or its context at some point in time. We assume that this sequence is generated by a causal connection layer—see Section 2.1.1. This layer has a clock whose time domain is the set of non-negative real numbers $\mathbb{R}_0^+$. The layer uses the clock to timestamp events. An element of the time domain is called a *time point*. Intuitively, the *history* h_τ of a system with respect to an event at time point τ is the sequence of all observed events in the behavior from its beginning, i.e., time point 0, up to and including τ. For brevity, we group all changes with the same time point in one event. However, we require that time in the history eventually diverges, thereby ruling out Zeno behaviors and ensuring that no event groups an infinite amount of changes.[1]

As mentioned previously, an architectural RTM represents a snapshot of the system architecture in terms of components and connectors. Figure 2.2 illustrates such a snapshot (see also Example 2.1.1), capturing a fragment of the `Architecture` instance, based on the metamodel in Figure 2.1. Owing to the causal connection, an event which occurs at time point τ and reflects a change in the system is mirrored in the RTM, effectively yielding a new version of the model. Therefore, the history h_τ of the system can be represented by a sequence comprising versions of RTM.

1 In the use-cases of interest, Zeno behaviors will not occur as, in practice, all examined histories will eventually be finite. Moreover, differences between measurements cannot become infinitely small.

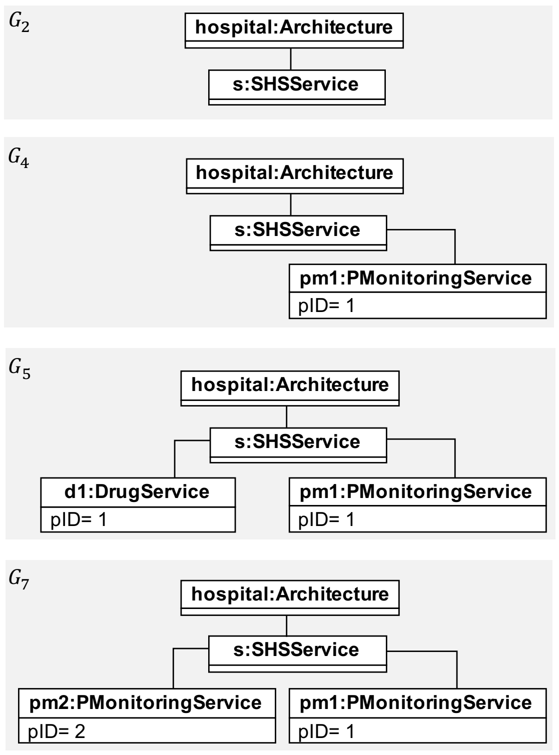


Figure 2.3: The history h_7^G comprising the RTMs G_2, G_4, G_5, and G_7

In this history representation based on RTMs, each member is an RTM G_τ which mirrors the events that occurred in the system at τ. For example, assume the history h_5, in which three events occur which correspond to the following changes to the RTM: at time point 2, a vertex `hospital` of type `Architecture`, the vertex `s` of type `SHSService`, and an association between the two vertices are added; at time point 4 a newly created vertex of type `PMonitoringService` with pID= 1 is associated with `s`; at time point 5, a newly created vertex of type `PDrugService` with pID= 1 is associated with `s`. Each change yielded an RTM version, therefore the history h_5 may be represented by a sequence $h_5^G := G_2G_4G_5$. The RTMs G_2,G_4, and G_5 are shown in Figure 2.3. Each RTM in the sequence is yielded by an event, is associated with the time point of the yielding event, and extends its predecessor by the changes corresponding to the event. The RTM G_2 extends the empty model $\varnothing$ which is assumed to be at the start of every such sequence.

In a history h_τ, τ is the time point of the last member of the sequence. The position of an RTM G_τ in a history comprising RTMs h_τ^G is given by an index $i \in \mathbb{N}^+$. We use the shorthand τ_i to denote the time point at position i. The spawning of an RTM based on an event, implies an assumed mapping from the set of events $\mathcal{E}$ to corresponding model modifications. For instance, according to the mapping, the event at time point 7 corresponds to: the addition of `pm2` and its association to `s`; the deletion of `d1` and its disassociation to `s`—when the event occurs, owing to causal connection, these modifications are applied to the RTM. To include the latest event, the history representation h_5^G is extended by the RTM G_7, i.e., $h_7^G = h_5^G \cdot G_7$—illustrated in Figure 2.3.

2.2 GRAPHS, GRAPH QUERIES, AND GRAPH TRANSFORMATION

An architectural RTM is amenable to a graph-based encoding, specifically a *typed attributed graph*, where entities are modeled as vertices, connectors as edges, and entity attributes as vertex attributes [156]. A typed attributed graph is *typed* over a *type graph* which defines types of vertices, edges, and attributes and valid structures for typed attributed graphs—similarly to the relationship between a metamodel and a model. Encoding an RTM as a graph allows for the realization of model transformation via established formalisms, such as *(typed attributed) Graph Transformation* (GT) [53], where GT *rules* are used to search for a part of the model which is transformed in place [67].

We employ sequences of typed attributed graphs to represent a history h_τ^G based on RTMs. Moreover, we employ graph queries to search a system snapshot for fragments of interest captured by graph patterns. Finally, we present previous work on an operationalization framework for graph queries based on graph transformation, which enables incremental query evaluation.

2.2.1 *Typed Attributed Graphs*

In the following, we introduce (typed) graphs and (typed) graph morphisms.

Definition 2.2.1 (graph, graph morphism)**.** A *graph* $G = (G^V, G^E, s^G, t^G)$ consists of a set of vertices G^V, a set of edges G^E, a source function $s^G : G^E \to G^V$, and a target function $t^G : G^E \to G^V$. Given two graphs $G = (G^V, G^E, s^G, t^G)$ and $K = (K^V, K^E, s^K, t^K)$, a *graph morphism* $f : G \to K$ is a pair of mappings $f^V : G^V \to K^V, f^E : G^E \to K^E$ such that $f^V \circ s^G = s^K \circ f^E$ and $f^V \circ t^G = t^K \circ f^E$.

A graph morphism $f : G \to K$ is a *monomorphism*, denoted by $\hookrightarrow$, if f^V and f^E are injective.

Definition 2.2.2 (typed graph, typed graph morphism)**.** A *type graph* is a distinguished graph $TG = (TG^V, TG^E, s^{TG}, t^{TG})$. A tuple $(G, type)$ consisting of a graph G and a graph morphism $type : G \to TG$ is called a *typed graph*. Given two typed graphs $G^T = (G, type)$ and $K^T = (K, type')$, a *typed graph morphism* $f : G^T \to K^T$ is a graph morphism $f' : G \to K$ such that $type' \circ f' = type$.

Typed graphs can be extended by vertex and edge attributes to obtain *typed attributed graphs* [53]. Our contributions rely on the typed, attributed graphs and graph morphisms introduced in our previous joint work [143, 144]. In the following, to avoid the complication of presentation, we omit attributes in technical representations of graphs and graph morphisms.

Attributes are associated with a data type, i.e., a character string, an integer, a real number, or a boolean. Graphs may contain a set of *attribute assignments* A which assign data-type-compatible values to attributes, e.g., $\mathtt{pm}_1.pID = 1$ in Figure 2.2. Formally, the set of attribute assignments A may have various representations, e.g., distinguished data vertices [53] or an attribute constraint over sorted variables [142].

The metamodel in Figure 2.1 may be seen as an informal representation of the type graph of the SHS, where only vertices have attributes. Correspondingly, the RTM in Figure 2.2 is an informal representation of a typed attributed graph. We henceforth refer to typed attributed graphs simply as graphs.

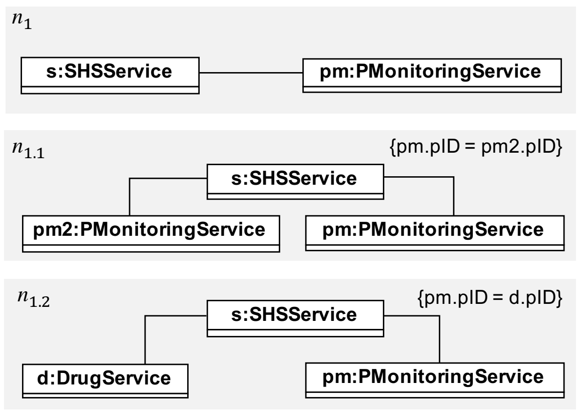


Figure 2.4: Statements on SHS as patterns—braces contain boolean expressions over the values of attributes of the pattern; vertices with the same label refer to the same vertex in the host graph

The history representation based on RTMs from Section 2.1.3 may be represented as a *timed graph sequence* [68], i.e., a sequence of graphs where: changes between two consecutive graphs are represented by morphisms describing additions and deletions; the timing of changes is computed based on the relative duration between the graphs.

2.2.2 *Graph Queries*

A *graph query* is the equivalent graph-based notion of a model query, i.e., in this context, a means to explore an existing graph. Typically, a simple graph query searches for a smaller graph, henceforth called a (graph) *pattern*, in the existing (queried) graph, the latter also called a *host graph*. The evaluation of a simple graph query over a host graph, also called (graph) *pattern matching*, amounts to finding *matches*, i.e., occurrences of the query pattern, in the host graph. Intuitively, a match is a mapping from the pattern to the host graph which preserves structure and type.

Definition 2.2.3 (simple graph query, graph query language, answer set, match)**.** Given a typed graph TG, a *simple graph query* θ is characterized by a graph pattern n typed over TG. A *graph query language* is a set of graph queries. The *answer set* $\mathcal{A}$ for θ over a host graph G is a set of all graph morphisms $m : n \hookrightarrow G$ typed over TG. An element of $\mathcal{A}$ is also called a *match*.

Besides the set of assignments A, a pattern may be extended with a set of *constraints* γ, i.e., boolean logical expressions over the values of attributes of matches. The attribute values of a match for a simple graph query characterized by such a pattern have to satisfy the assignments and constraints in A and γ of the pattern, respectively. Examples of patterns based on the SHS are shown in Figure 2.4, where the γ in patterns is visualized between braces.

In certain cases, simple graph queries, i.e., queries searching only for a pattern, are not sufficient as a language for expressing more complex queries, for instance, patterns whose existence in the host graph should be prohibited or disconnected patterns which should simultaneously exist in the host graph.

In these cases, a simple graph query may be enhanced with an *Application Condition* (AC), i.e., a condition which every match m should satisfy. Then, a graph query θ is characterized by a pattern n and an AC ac over n, and denoted by $\theta \coloneqq (n, ac)$.

The language of *Nested Graph Conditions* (NGCs) [75] can formulate ACs that are as expressive as first-order logic on graphs [38] as shown in [75, 124] and constitutes, as such, a natural formal foundation for pattern-based queries.

Definition 2.2.4 (nested graph conditions)**.** Let n,$\hat{n}$ be patterns. Then ϕ is a *nested graph condition* (NGC) over n defined as follows.

$$\phi_n ::= true \mid \neg\phi_n \mid \phi_n \wedge \phi_n \mid \exists(f : n \hookrightarrow \hat{n}, \phi_{\hat{n}})$$

The morphism f in the existential quantifier *binds*, i.e., relates, elements in patterns of outer conditions (n) to patterns of inner (nested) conditions ($\hat{n}$) and is therefore also called a *binding*. In the remainder, we abbreviate $\exists(f, true)$ by $\exists f$ and, when the domain of f is clear from context, $\exists(f : n \hookrightarrow \hat{n}, \phi_{\hat{n}})$ by $\exists(\hat{n}, \phi)$. Moreover, abbreviations such as disjunction ($\vee$), and the universal quantifier ($\forall$) can be defined as usual.

NGCs can be extended with typing over a given type graph TG as usual [53] by adding typing morphisms from each graph to TG and by requiring type-compatibility with respect to TG for each graph morphism.

The semantics of NGCs are given by the satisfaction relation below.

Definition 2.2.5 (satisfaction of nested graph conditions)**.** Let G be a host graph, n a pattern, and $m : n \hookrightarrow G$ a morphism. Moreover, let ϕ be an NGC over n. Then m satisfies ϕ, written $m \vDash_{NGC} \phi$, if one of the following items applies.

- $\phi = true$.
- $\phi = \neg\phi'$ and $m \nvDash_{NGC} \phi'$.
- $\phi = \phi' \wedge \phi''$, $m \vDash_{NGC} \phi'$, and $m \vDash_{NGC} \phi''$.
- $\phi = \exists(f : n \hookrightarrow \hat{n}, \phi')$ and there exists $\hat{m} : \hat{n} \hookrightarrow G$ with $\hat{m} \circ f = m$ and $\hat{m} \vDash_{NGC} \phi'$.

Intuitively, the existential quantifier in a query $(n, \exists(n \hookrightarrow \hat{n}, \hat{\phi}))$ is satisfied for a match m for n when (i) there exists a match $\hat{m}$ for $\hat{n}$ in G such that $\hat{m}$ satisfies $\hat{\phi}$ (ii) $\hat{m}$ is *compatible* with m, i.e., respects the binding between the two patterns captured in $n \hookrightarrow \hat{n}$. The operator *true* is always satisfied. The intuition behind negation and conjunction is similar to that in first-order logic.

Let $\mathcal{L}$ be the query language of queries with AC based on NGCs and $\theta \coloneqq (n, \phi)$ a query in in $\mathcal{L}$, i.e., ϕ is an NGC over n. The answer set $\mathcal{A}$ for θ over a host graph G contains all matches m in G for the query pattern n such that $m \vDash_{NGC} \phi$. We also denote $\mathcal{A}$ by $\mathcal{A}(G)$ when θ is clear from context.

Example 2.2.1 (query with a nested graph condition as an application condition)**.** *Assume the following hypothetical requirement which draws from operation sequence compliance, i.e., the order of service invocations [see 161], in an SHS: "When a sensor service is invoked for a patient by the main service,* *there exists no other sensor service for the same patient. Moreover, a drug service should be invoked for the same patient." Based on the SHS metamodel in Figure 2.1, the main service is represented by* `SHSService`*, the sensor service by* `PMonitoringService`*, and the drug service by* `DrugService`*. Then, the described situations, i.e., sensor service invoked by a main service, may be captured by the patterns n_1, $n_{1.1}$, and $n_{1.2}$, illustrated in Figure 2.4, where the attribute constraints in $n_{1.1}$ and $n_{1.2}$ (illustrated between braces) ensure that the situations concern the same patient.*

Formulated as a query in $\mathcal{L}$, the requirement is then translated into: Find all matches of pattern n_1 in G that satisfy ϕ_1, i.e., where a match for $n_{1.1}$ does not exist while a match for $n_{1.2}$ does. In $\mathcal{L}$, this query is captured by $\theta_1 := (n_1, \phi_1)$ with ϕ_1 an NGC defined as $\neg(\exists(n_1 \hookrightarrow n_{1.1}, \text{true})) \wedge (\exists(n_1 \hookrightarrow n_{1.2}, \text{true}))$, or, abbreviated, $\neg(\exists n_{1.1}) \wedge \exists n_{1.2}$. Nesting implies that vertices `s` *and* `pm` *from n_1 are* bound *in inner patterns $n_{1.1}$ and $n_{1.2}$, i.e., all patterns refer to the same* `s` *and* `pm` *in G. In our illustrations this is encoded by the usage of the same label for bound elements.*

The $\mathcal{A}(G_5)$ for θ_1, where G_5 is illustrated in Figure 2.3, consists of one match for n_1 which satisfies ϕ_1, i.e., a `pm` *is found which is connected to a* `DrugService` *with the same* pID *and not connected to any other sensor services.*

2.2.3 *Graph Transformation*

We briefly present the well-established algebraic approach to typed attributed *Graph Transformation* (GT) [53], where the transformation is performed based on GT rules. Let G be a graph (in this context, encoding an RTM) typed over a type graph TG. Then, $r := \langle f : n \hookrightarrow R \rangle$ is a *simple GT rule*, characterized by a *Left-hand Side* (LHS) graph (n) and a *Right-hand Side* (RHS) graph (R)—also typed over TG. The LHS and RHS define the pre-condition and post-condition of an application of r, respectively. Intuitively, the application of r searches for occurrences of the LHS in G, and transforms G according to the RHS. Similarly to queries, simple rules can be extended with ACs, i.e., $r := \langle f : n \hookrightarrow R, ac_n \rangle$, thereby requiring that occurrences of the LHS satisfy the condition ac. Rules may be combined to form a pair that describes post-conditions which perform both insertions and deletions in a single rule application in G. A set of GT rules typed over TG is called a (typed) *Graph Transformation System* (GTS).

Amalgamated GT rules [25, 69] allow for matching various graph structures and *amalgamating*, i.e., synchronizing, these matches over a common kernel match. An amalgamated rule is defined based on an *interaction scheme* which comprises a *kernel rule* and *multi-rules*. The kernel rule contains elements common to all rules and is to be applied only once; a multi-rule extends the kernel rule and may be applied arbitrarily many times. The interaction scheme also contains *kernel morphisms* which embed the kernel rule into each multi-rule. An amalgamated rule is constructed via the application of the interaction scheme on a host graph: For each match for a multi-rule in the graph which overlaps in a match for the kernel rule, a copy of the multi-rule is created together with a kernel morphism from the kernel rule; all copies are subsequently *glued* at their common kernel rule which induces the amalgamated rule. A more elaborate description of amalgamated GT rules is presented in Section A.2.

2.2.4 *Query Operationalization via Graph Transformation*

A graph query is a declarative means to express a structure of interest which should satisfy a given condition. The query itself does not provide instructions on how it is to be evaluated, i.e., its *operationalization*. Operationalization typically comprises the decomposition of a query into a *network*, i.e., a suitable ordering, of simpler sub-queries, e.g., the query evaluation engine in [155] which is capable of decomposing a query in $\mathcal{L}$ into a *RETE* network based on the eponymous algorithm [58]. For formally underpinning the operationalization of temporal queries, we build on the formal framework presented in [23] which also supports queries in $\mathcal{L}$. The framework in question decomposes a query with an arbitrarily complex NGC as AC into a generalized form of RETE networks called *Generalized Discrimination Network* (GDN) [79].

A GDN is a directed acyclic graph where each node represents a (sub-)query. To avoid confusion, we refer to the GDN by network. Dependencies between queries are represented by edges from child nodes, i.e., the nodes whose results are required, to the parent node, i.e., the node which requires the results. Dependencies can either be *positive*, i.e. the query represented by the parent node requires the presence of matches of the child node, or *negative*, i.e., the query of the parent node forbids the presence of such matches. The overall query is evaluated bottom-up: The execution of the network starts with leaves and proceeds upwards. The terminal node computes the $\mathcal{A}$ of the query.

The framework in [23] employs typed GT as a basis, and therefore formally underpins the specification and operationalization of graph queries based on GT rules. Specifically, the framework realizes a GDN as a GTS with non-deleting GT rules. Given a query $(n, \phi) \in \mathcal{L}$, the query pattern n and each NGC operator in ϕ induce a GDN node, i.e., a (sub-)query, which in turn corresponds to a GT rule. The LHS of the rule realizes a query that searches for matches in a given host graph G. The RHS of the rule creates a *marking node* in G that *marks* each match and *marking edges* from the marking node to each node of the match—marking nodes are not to be confused with regular graph nodes in G (which, in this context, represent entities of the modeled system), thus we use the term *vertex* for regular graph nodes. In order to be able to create marking nodes and edges, the GT rules of a GDN, henceforth called *marking rules*, are typed over an extended type graph TG' which adds the required types for marking nodes and edges to the initial type graph TG. The LHS of rules with dependencies have a (non-nested) AC that require the existence of marking nodes of their positive dependencies and forbid the existence of marking nodes of their negative dependencies.

Example 2.2.2 (generalized discrimination network)**.** *The* Generalized Discrimination Network *(GDN) for θ_1 from Example 2.2.1 is shown in Figure 2.5 (right), where each square represents a GDN node.*

The GDN consists of five nodes. All nodes represent marking rules whose LHS searches for matches of a pattern and whose RHS creates marking nodes and edges that mark the matches found by the LHS. The node $r_{n_{1.1}}$ searches for the pattern $n_{1.1}$ and the node $r_{n_{1.2}}$ searches for the pattern $n_{1.2}$. Although technically the nodes $r_{n_{1.1}}$ and $r_{n_{1.2}}$ depend on marking nodes created by the marking rules induced for true*, in practice marking rules for* true *are omitted as their dependency is*

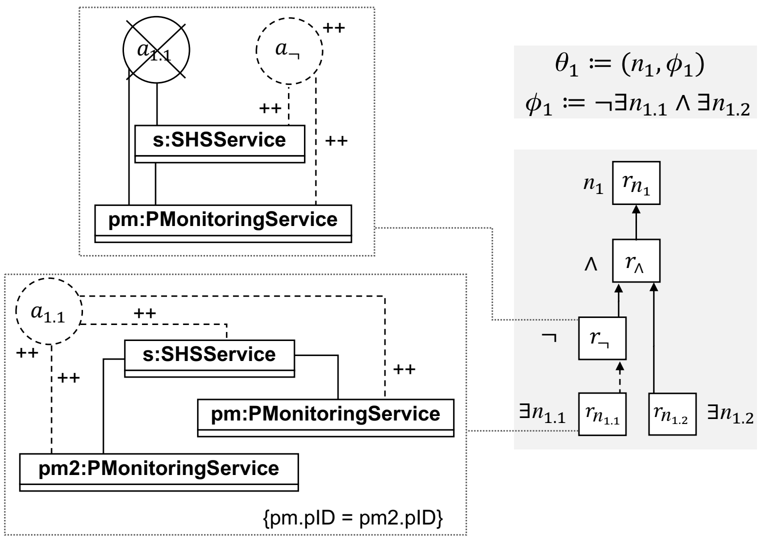


Figure 2.5: GDN (right) and the marking rules $r_{n_{1.1}}$, $r_{\neg}$ for θ_1

always satisfied. The node $r_{\neg}$ is the parent of a negative dependency (drawn with a dashed line), i.e., the AC of $r_{\neg}$ is satisfied when a marking node created by r_{n_1} does not *exist. The AC of the node $r_{\wedge}$ is satisfied when both of the dependencies of the marking rule are satisfied. The LHS of the nodes induced by conjunction and negation is inherited from their parent node; in this example the LHS of $r_{\wedge}$ and $r_{\neg}$ is the pattern n_1. Finally, the topmost node r_{n_1} searches for matches for the pattern n_1, and returns those matches that satisfy its dependency, i.e., the AC ϕ_1.*

The marking rules for nodes $r_{n_{1.1}}$ and $r_{\neg}$ are shown in Figure 2.5 (left—within rectangles), where (i) marking nodes are illustrated by circles, (ii) forbidden marking nodes are crossed, and (iii) the marking nodes and marking edges added by a marking rule are dashed and annotated with "++". The figure shows a compact view where the illustrations of rules contain both their LHS (patterns) and RHS (marking nodes and edges).

The technical representation of a marking rule of a GDN is as follows—based on notation and functions from Definition 2.2.1 and the GDN from Example 2.2.2. The rule $r_{n_{1.1}}$ is given by $\langle p_{n_{1.1}} : n_{1.1} \hookrightarrow R, \neg\exists p_{n_{1.1}} \rangle$ with the RHS R of the rule entailing (i) the creation of a marking node $a_{n_{1.1}}$, i.e., $R^V = n_{1.1}^V \uplus a_{n_{1.1}}$ (ii) and the creation of edges from the marking node to all marked vertices, i.e., $R^E = n_{1.1}^E \uplus \{b_k | k \in n_{1.1}^V\}$ such that $s^R(b_k) = a_{n_{1.1}}$ and $t^R(b_k) = k$. The marking node and marking edges are typed over dedicated types that each rule induces in the extended type graph TG'. The AC of the rule, i.e., $\neg\exists p_{n_{1.1}}$, requires that a match for the LHS of $r_{n_{1.1}}$ is not marked more than once.

The rule $r_{\neg}$ is given by $\langle p_{\neg} : n_1 \hookrightarrow R', \neg\exists p_{\neg} \wedge \neg\exists p_{n_{1.1}} \rangle$. The LHS of nodes induced by negation and conjunction is identical to their *context pattern*, i.e., the pattern of their parent node; therefore the LHS of $r_{\wedge}$ is the same as n_1 and so is the LHS of $r_{\neg}$. Regarding the RHS, $R'^V = n_1^V \uplus a_{n_{\neg}}$ and $R'^E = n_1^E \uplus \{b_k | k \in n_1^V\}$ such that $s^{R'}(b_k) = a_{\neg}$ and $t^{R'}(b_k) = k$. The marking nodes and edges are typed over dedicated types in TG'. Besides the requirement for a single marking node per match, i.e., $\neg\exists p_{\neg}$, the AC of $r_{\neg}$ further prohibits the existence of a marking node for its dependency $r_{1.1}$, i.e., $\neg\exists p_{n_{1.1}}$.

2.2.5 *Local Search and Incremental Evaluation*

Optimization techniques such as *local search* can be employed to reduce the pattern matching effort of GDN nodes [see 5]. In local search, pattern matching initiates from a single element and builds a match candidate iteratively following a heuristics-based search plan.

Owing to the decomposition of the query into simpler marking rules, a network such as the GDN or RETE, is amenable to *incremental execution*. Changes in a host graph G are propagated through the network, whose nodes only recompute their results if the change concerns them or one of their dependencies. Therefore, we say that the query is also *evaluated incrementally* as its answer set $\mathcal{A}$ is updated by each network execution. If a re-computation is deemed necessary, owing to local search, a node is capable of updating its matches starting from changed elements instead of starting over.

The previously mentioned RETE-based engine in [155] combines incremental query evaluation with local search. The engine has been shown to be adequately fast in evaluating structural queries over significantly large graphs—typically representing RTMs, where, similarly to the adaptation setting, the query is evaluated *reactively*, i.e., after every change to the model. The effectiveness of this engine indicates that the combination of local search and incremental evaluation is also a solid foundation for the evaluation of temporal queries.

2.3 METRIC TEMPORAL GRAPH LOGIC

Adding a temporal dimension to the exemplary requirement from Example 2.2.1 makes it similar to compliance checking of medical procedures which may track time between triage and admission [108], here represented by the invocation of a sensor service (captured in the pattern n_1) and a drug service ($n_{1.2}$), respectively: "*When a sensor service is invoked for a patient, there should be a drug service invoked for the same patient within one minute and, until then, there should be no other sensor service invoked for the same patient.*" The specific timing constraint is adjusted for the purpose of presentation.

Formulated as a query, this instruction introduces a quantitative temporal requirement on the evolution of graph structures: "find all matches for n_1, such that a match for n_1 at a time point τ, at least one match for $n_{1.2}$ is found at some time point $\tau' \in [\tau, \tau + 60]$, i.e., at most 60 seconds later; in addition, at each time point $\tau'' \in [\tau, \tau')$ in between, no match for $n_{1.1}$ is present", where all n patterns refer to the same patient, i.e., the elements `s` and `pm` are common in all matches, and time is assumed to be tracked in seconds.

The query language $\mathcal{L}$ introduced in Section 2.2, which employs NGCs for the definition of AC, does not inherently support temporal requirements. Quantitative temporal requirements are typically specified in metric temporal logics, such as the *Metric Temporal Logic* (MTL) [96]. MTL is defined over Kripke structures and represents a system state by a subset of a finite set of atomic propositions. Thus, MTL does not support the use of bindings of elements in graphs to express how a certain match evolves over the system evolution, i.e., in a sequence of graphs. Therefore, for a sensor service invocation, i.e., a match m_1 for n_1, and a drug service invocation, i.e., a match $m_{1.2}$ for $n_{1.2}$, MTL is unable to ensure they refer to the same `s` and `pm`, and thus check whether the invocations concern the same patient.

To enable the definition and analysis of graph conditions with quantitative temporal requirements on the evolution of patterns, in our previous joint work we presented the *Metric Temporal Graph Logic* (MTGL) [68, 143]. MTGL builds on NGCs and MTL. MTGL is defined over graph conditions, called *Metric Temporal Graph Conditions* (MTGCs). The logic employs the support for bindings from NGCs and, therefore, is able to track the evolution of a given match in a sequence of graphs separately to other matches. We focus on a subset of MTGL operators which, besides NGC operators introduced in Definition 2.2.4, contains the *metric*, i.e., interval-based, temporal operators *until* (U_I, with I an interval in $\mathbb{R}_0^+$) and its dual *since* (S_I) from MTL.

Definition 2.3.1 (metric temporal graph conditions)**.** Let n,$\hat{n}$ be patterns and $f : n \hookrightarrow \hat{n}$ a binding. Moreover, let I be an interval in $\mathbb{R}_0^+$. Then ψ is a *Metric Temporal Graph Condition* (MTGC) over n defined as follows.

$$\psi_n ::= \mathit{true} \mid \neg\psi_n \mid \psi_n \wedge \psi_n \mid \exists(f : n \hookrightarrow \hat{n}, \psi_{\hat{n}}) \mid \psi_n \mathrm{U}_I \psi_n \mid \psi_n \mathrm{S}_I \psi_n$$

The operators *eventually* ($\lozenge_I$) and *once* ($\blacklozenge_I$) are abbreviations of *until* and *since*: $\lozenge_I \psi = \mathit{true}\,\mathrm{U}_I\,\psi$ and $\blacklozenge_I \psi = \mathit{true}\,\mathrm{S}_I\,\psi$. Abbreviations for NGCs (see Definition 2.2.4) may also be used.

In MTGL, the AC in the example above, i.e., "given a match for n_1 at a time point τ, at least one match for $n_{1.2}$ is found at some time point $\tau' \in [\tau, \tau+60]$, i.e., at most 60 seconds later; in addition, at each time point $\tau'' \in [\tau, \tau')$ in between, no match for $n_{1.1}$ is present", is captured by the MTGC $\psi_1 := \neg\exists(n_1 \hookrightarrow n_{1.1}, \mathit{true})\,\mathrm{U}_{[0,60]}\,\exists(n_1 \hookrightarrow n_{1.2}, \mathit{true})$, or, abbreviated, $\neg\exists n_{1.1}\,\mathrm{U}_{[0,60]}\,\exists n_{1.2}$.

MTGL reasons over (finite) timed graph sequences (see Section 2.2.1) which capture the history of the system state—similarly to the RTM sequence in Figure 2.3 where each member of the sequence represents a state of the architecture. However, MTGCs can also be equivalently checked over a *graph with history* [68, 143] defined as follows.

Definition 2.3.2 (graph with history)**.** Let TG be a type graph where all vertices and edges have the attribute *cts*, denoting the time point of their creation, and *dts*, denoting the time point of their deletion. Then H is a *graph with history* if H is typed over TG and the following conditions hold:

- The set of attribute assignments A of H contains assignments $\mathit{cts} = x$ and $\mathit{dts} = y$ for each vertex and each edge.
- $x \geq 0$ and either $y > x$ or, if the element has not been deleted, $y = -1$.
- For an edge d in H:
  - the value of the *cts* attributes of the source and target vertices of d are less or equal to the *cts* value of d, i.e., $(s^H(d)).\mathit{cts} \leq d.\mathit{cts}$ and $(t^H(d)).\mathit{cts} \leq d.\mathit{cts}$.
  - the value of the *dts* attributes of the source and target vertices of d are greater or equal to the *dts* value of d, i.e., $(s^H(d)).\mathit{dts} \geq d.\mathit{dts}$ and $(t^H(d)).\mathit{dts} \geq d.\mathit{dts}$.

Once set, the assignments in A for all attributes in H are fixed; the only exception is *dts*, whose value may change once, i.e., from -1 to another value.

Intuitively, in a graph with history H all elements must have a *cts* and *dts* to which a value has been assigned; when an element is created, a value is also assigned to its *dts*, this value being -1; this value changes when the element is deleted in the modeled system. As an element cannot have been deleted prior to its creation or deleted simultaneously with its creation, the value of *dts*, if not -1, has to be larger than the value of *cts*. Except for the *dts*, all other values of attributes in H (including *cts*) are fixed, once set. Finally, edges in H must have been created at a time point when both of their endpoints exist, and must have been deleted at a time point when one of its endpoints does not exist.

A graph with history can be obtained by *folding* a timed graph sequence into a single graph that satisfies the conditions above [68, 143].

In the following we define the semantics of the satisfaction relation of MTGL based on graphs with history.

Definition 2.3.3 (satisfaction of metric temporal graph conditions over a graph with history)**.** Let H be a graph with history, n a pattern, and $m : n \hookrightarrow H$ a match. Moreover, let τ be a time point in $\mathbb{R}_0^+$ and ψ be an MTGC over n. Then m satisfies ψ at τ, written $(m, \tau) \models \psi$, if $\max_{e \in E} e.cts \leq \tau < \min_{e \in E} e.dts$, with E the elements of m, and one of the following cases applies.

- $\psi = \mathit{true}$.
- $\psi = \neg\chi$ and $(m, \tau) \not\models \chi$.
- $\psi = \chi \wedge \omega$, $(m, \tau) \models \chi$, and $(m, \tau) \models \omega$.
- $\psi = \exists(f : n \hookrightarrow \hat{n}, \chi)$ and there exists $\hat{m} : \hat{n} \hookrightarrow H$ such that $\hat{m} \circ f = m$ and $(\hat{m}, \tau) \models \chi$.
- $\psi = \chi \mathrm{U}_I \omega$ and there exists τ' with $\tau' - \tau \in I$ such that $(m, \tau') \models \omega$ and for all $\tau'' \in [\tau, \tau')$ $(m, \tau'') \models \chi$.
- $\psi = \chi \mathrm{S}_I \omega$ and there exists τ' with $\tau - \tau' \in I$ such that $(m, \tau') \models \omega$ and for all $\tau'' \in (\tau', \tau]$ $(m, \tau'') \models \chi$.

Intuitively, a match m for n in the graph with history H satisfies the MTGC $\exists(f : n \hookrightarrow \hat{n}, \chi)$ at time point τ if (i) all elements of m are already created but not yet deleted at τ, and (ii) there exists a compatible match $\hat{m}$ for $\hat{n}$ in H such that $\hat{m}$ satisfies the MTGC χ. The operator *until* is satisfied by a match m at time point τ when there is a future time point τ' within the operator interval I such that m satisfies ω and, until then, i.e., for all time points between τ and τ', m continuously satisfies χ. The intuition is inverted for the operator *since*, which is satisfied by a match m at time point τ when there is a past time point τ' within the operator interval I such that m satisfies ω and, since then, i.e., for all time points between τ' and τ, m continuously satisfies χ.

3 RUNTIME MODEL WITH HISTORY AND SPECIFICATION OF TEMPORAL QUERIES

This chapter presents a compact encoding of a sequence of RTMs called *Runtime Model with History* (RTM$^{\mathrm{H}}$), which simultaneously provides a view of the latest state of the architecture as well as of the entire history, i.e., previous snapshots, of the state. The RTM$^{\mathrm{H}}$ is described in detail in Section 3.1.

Based on the RTM$^{\mathrm{H}}$, we introduce in Section 3.2 a query language for the specification of temporal model queries with quantitative timing constraints. The language supports the formulation of temporal requirements as formulas in the interval-based temporal logic introduced in Section 2.3. The query answre set over a given RTM$^{\mathrm{H}}$ comprise those matches for the query pattern in the model whose evolution satisfies the formula. Relying on the history encoding in the RTM$^{\mathrm{H}}$ and the capability of interval-based logics to check satisfaction for the entire time domain, we equip answers with timing information. Specifically, we pair each match with its *temporal validity*, i.e., the period for which the match exists in the RTM$^{\mathrm{H}}$ *and* satisfies the formula.

As the given RTM$^{\mathrm{H}}$ may encode a history which may be unfinished, i.e., an ongoing system execution, future changes to the RTM$^{\mathrm{H}}$ may affect the temporal validity of a match. Section 3.3 presents an additional answer set type which relies on a novel three-valued interpretation of the logic and ensures that the temporal validity of a match only includes time points for which the satisfaction decision will not be affected by future changes to the RTM$^{\mathrm{H}}$.

3.1 RUNTIME MODEL WITH HISTORY

A *Runtime Model with History* (RTM$^{\mathrm{H}}$) is an enhanced architectural RTM that simultaneously provides two views on the modeled system: a view of the *current* system state of the architecture, which corresponds to a conventional causally-connected RTM; and a compact view of the *history* of the state. Formally, the RTM$^{\mathrm{H}}$ is founded on graphs with history—see Definition 2.3.2. The view on history in an RTM$^{\mathrm{H}}$ is afforded by each entity being equipped with a *creation* timestamp and a *deletion* timestamp, abbreviated *cts* and *dts*, respectively. For an example, see Figure 2.1 where all entities inherit from the `MonitorableEntity`.

The *cts* and *dts* capture the time points of creation and deletion of an entity, respectively. Upon the occurrence of an event in the system, similarly to an event yielding a new system state, the corresponding entity creation (deletion) in the RTM$^{\mathrm{H}}$ yields a new instance of the RTM$^{\mathrm{H}}$ where the *cts* (*dts*) of the modified entity is set based on the time point of the event. When an entity is created, its *dts* is set to ∞—instead of -1, which is the default assignment in a graph with history. When an entity is deleted in the modeled system, the respective entity is *not* deleted in the RTM$^{\mathrm{H}}$. Rather, its *dts* is updated to the time point of the event that induced the deletion. Therefore, the value of the *dts* of an entity in an RTM$^{\mathrm{H}}$ is always larger than the value of the *cts* for the same entity. For an entity e, we define its *lifespan* as the non-empty non-negative

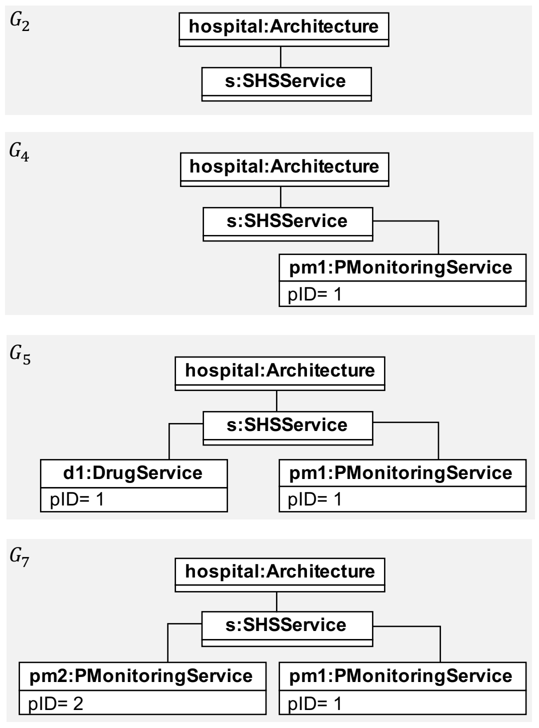


Figure 3.1: The history h_7^G comprising the RTMs G_2, G_4, G_5, and G_7

interval $[e.cts, e.dts)$. The intuition behind the lifespan of an entity being right-open[1] is that if e has been deleted at a time point τ this means that e has existed until a time point that approaches but is not equal to τ.

An RTM$^{\mathrm{H}}$ requires that connectors exist for as long as both their end-points exist. Moreover, an RTM$^{\mathrm{H}}$ has no connector attributes, i.e., edge attributes, and attribute values of entities may only be set once when the entities are created—the value of the *dts* being the only exception; these characteristics incur no loss of generality as, if changes to connectors or attribute values are of interest, an RTM$^{\mathrm{H}}$ can track these changes by modeling these elements as entities in the metamodel—as shown in [97]. If an attribute value is modeled as an entity in the metamodel, each value change would then lead to a creation of a new entity in the RTM$^{\mathrm{H}}$, where the duration of the value would be captured by a *cts* and a *dts*. Similarly, modeling a connector as an entity allows for supporting connector attributes and tracking changes to connectors—we demonstrate this modeling technique in the evaluation in Section 5.3.

By retaining all entities as well as information on their creation and deletion time points, an RTM$^{\mathrm{H}}$ instance suffices to represent the evolution of entities up to and including the time point of the event that yielded the instance in question. In contrast to a sequence of RTMs, which stores multiple RTMs and all their entities to be able to track the evolution of entities over the sequence, an RTM$^{\mathrm{H}}$ stores only a single instance of each model entity. Hence, it affords a compact and, therefore, more memory-efficient encoding of history. Similarly to an RTM in a sequence, an instance of the RTM$^{\mathrm{H}}$ is associated with the time point

1 An interval is right-open when its right end-point is excluded from the interval. See Section A.1 for a comprehensive definition of intervals.

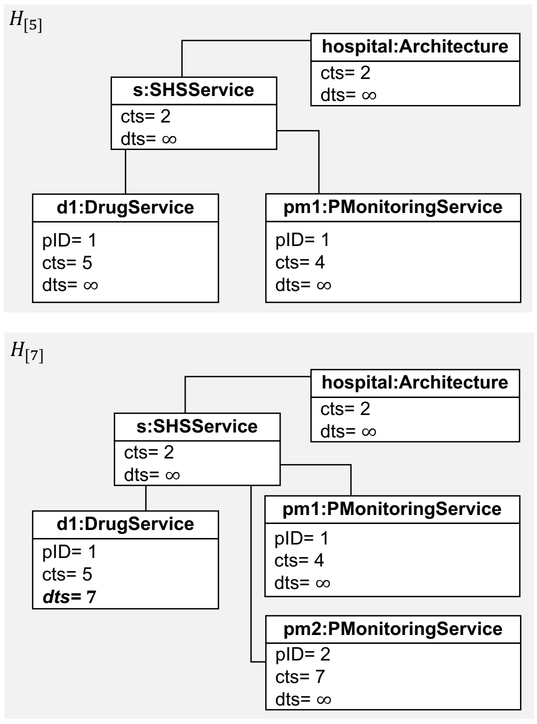


Figure 3.2: RTM$^{\mathrm{H}}$ Instances $H_{[5]}$ and $H_{[7]}$

of the member of the RTM sequence to which the instance corresponds. For example, the representation of h_5^G from Section 2.1.3 by an RTM$^{\mathrm{H}}$ yields a single model, $H_{[5]}$, that contains the same information as h_5^G and covers the same time period, i.e., $[0, 5]$. $H_{[5]}$ is illustrated in Figure 3.2 and, for convenience, h_7^G, which contains h_5^G, is illustrated again in Figure 3.1. We note that in an RTM$^{\mathrm{H}}$ causal connection only applies to the latest snapshot of the system state.

Technically, an RTM$^{\mathrm{H}}$ corresponding to a new event is obtained by a *coalescence* of the new event and the previous RTM$^{\mathrm{H}}$ instance into a new RTM$^{\mathrm{H}}$ instance. For creations, the new instance contains new entities corresponding to the event and sets the values of their attributes: regular attributes are set according to data in the event, the *cts* is set based on the time point of the event, and the *dts* is set to ∞. For deletions, the *dts* values of the affected entities are updated to the time point of the event. For example, when h_5 is incremented to h_7 by the event at time point evaluate the query, which corresponds to the creation of `pm2` and the deletion of `d1`, this event yields a new RTM$^{\mathrm{H}}$ $H_{[7]}$ which constitutes a coalescence of the event and $H_{[5]}$. $H_{[7]}$ is illustrated in Figure 3.2—`d1`.*dts* has been updated from the previous RTM$^{\mathrm{H}}$ and is accentuated.

Each event increments the history and spawns a new RTM$^{\mathrm{H}}$ instance, effectively yielding a sequence h^H comprising instances of an RTM$^{\mathrm{H}}$. However, entities are never deleted in an RTM$^{\mathrm{H}}$ instance, and attribute value assignments (except the *dts*) are fixed; thus, an RTM$^{\mathrm{H}}$ $H_{[\tau_i]}$, corresponding to the time point τ of the event with index i in the history, includes all entities and important timing information from $H_{[\tau_{i-1}]}$, rendering the storage of $H_{[\tau_{i-1}]}$ after the coalescence with the event at τ_i unnecessary—as demonstrated with $H_{[5]}$ and $H_{[7]}$ in Figure 3.2.

3.2 QUERY LANGUAGE

We introduce the query language $\mathcal{L}_{\mathrm{T}}$ which enables the specification of temporal model queries over an RTM$^{\mathrm{H}}$ via *temporal graph queries*. A temporal query $\zeta \in \mathcal{L}_{\mathrm{T}}$ is characterized similarly to graph queries with NGCs in $\mathcal{L}$ (see Section 2.2.2), i.e., $\zeta := (n, ac)$. However, in contrast to $\mathcal{L}$, temporal queries feature ACs based on MTGCs—see Section 2.3. MTGCs build on NGCs and allow for the specification of a desired ordering and timing constraints on the evolution of graph structures, thereby supporting temporal requirements.

In $\mathcal{L}_{\mathrm{T}}$, the query from Section 2.3, i.e., "find all matches for n_1, such that a match for n_1 at a time point τ, at least one match for $n_{1.2}$ is found at some time point $\tau' \in [\tau, \tau + 60]$, i.e., at most 60 seconds later; in addition, at each time point $\tau'' \in [\tau, \tau')$ in between, no match for $n_{1.1}$ is present", is captured by $\zeta_1 := (n_1, \psi_1)$ where ψ_1 is an MTGC defined as $\neg\exists\, n_{1.1}\, \mathrm{U}_{[0,60]}\, \exists\, n_{1.2}$. Recall that elements common to n_1 and the patterns $n_{1.1}, n_{1.2}$ are bound—see Section 2.2.

Similarly to entities in an RTM$^{\mathrm{H}}$ having lifespans, a match consisting of such entities also has a lifespan, defined as the set of time points for which matched entities coexist in the RTM$^{\mathrm{H}}$ and satisfy the attribute constraint. As discussed in Section 1.2, in a time-aware setting, the answer to whether a match satisfies its AC must be accordingly accompanied by timing information. Temporal logics that reason over intervals, such as MTGL, are capable of defining the truth value of a formula for every time point in the time domain. Building on this capability, compared to the answer set $\mathcal{A}$ for a query in $\mathcal{L}$, an answer set $\mathcal{T}$ for a query $\zeta \in \mathcal{L}_{\mathrm{T}}$ is extended with a temporal dimension: Matches in $\mathcal{T}$ are paired with a *temporal validity*, i.e., the intersection of their lifespan with the period for which the match satisfies the AC of ζ, called *satisfaction span*. We elaborate on the lifespan, the satisfaction span, and the temporal validity below.

3.2.1 *Lifespan of a Match*

Entities of an RTM$^{\mathrm{H}}$ have attributes which capture their creation (*cts*) and deletion (*dts*) timestamps. These attributes define the lifespan of an entity—see Section 3.1. A match m in an RTM$^{\mathrm{H}}$ $H_{[c]}$, with $c \in \mathbb{R}_0^+$, is valid only if there is a non-empty interval λ^m, called the *lifespan of the match*, during which the lifespans of all matched entities E overlap:

$$\lambda^m = \bigcap_{e \in E} [e.cts, e.dts) \tag{3.1}$$

Attribute values of matched entities do not change and, hence, cannot affect the lifespan computation. Entity timestamps are always assigned from $\mathbb{R}_0^+$, therefore it always holds that $\lambda^m \subseteq \mathbb{R}_0^+$. In the special case where the pattern of a query is the empty graph $\varnothing$, an (empty) match m is always found with $\lambda^m = \mathbb{R}_0^+$.

3.2.2 *Satisfaction Span and Temporal Validity*

We call the set of time points for which an MTGC is satisfied its *satisfaction span*, denoted by $\mathcal{Y}$. Given a query $(n, \psi) \in \mathcal{L}_{\mathrm{T}}$ or a nested condition with n as enclosing pattern, and a match m for n in the RTM$^{\mathrm{H}}$ $H_{[c]}$, the satisfaction span of m for ψ over $H_{[c]}$ is defined as $\mathcal{Y}(m, \psi) = \{\tau \mid \tau \in \mathbb{R} \wedge m \text{ satisfies } \psi \text{ at } \tau\}$.

The *temporal validity* of the match is the set of time points for which m exists in $H_{[c]}$ and satisfies ψ, i.e., the intersection of the lifespan of a match with the satisfaction span, and is denoted by $\mathcal{V}(m,\psi)$.

The intersection of two intervals is always an interval, whereas the union of two intervals may result in disjoint, i.e., disconnected, sets. To encode such unions, we define an *interval set* $\mathbb{I} \subseteq \mathbb{R}$ which may contain disjoint or empty intervals. Note that a set operation between an $\mathbb{I} \in \mathcal{F}$ and $I \in \mathfrak{I}$ with $\mathcal{F}$ and $\mathfrak{I}$ the set of all interval sets and intervals, respectively, may result into an $\mathbb{I}' \in \mathcal{F}$. The satisfaction span $\mathcal{Y}$ and the temporal validity $\mathcal{V}$ may depend on unions of intervals or operations with other interval sets and are, therefore, interval sets themselves.

The definition below presents the recursively defined *satisfaction computation* $\mathcal{Z}$ of an MTGC. An explanation of the intuition behind the definition follows.

Definition 3.2.1 (satisfaction computation $\mathcal{Z}$). Let $n, \hat{n}$ be patterns and ψ, χ, ω be MTGCs. Moreover, let m be a match for n in an RTM$^{\mathrm{H}}$ $H_{[c]}$. The *satisfaction computation* $\mathcal{Z}(m,\psi)$ is recursively defined as follows.

$$\mathcal{Z}(m, true) = \mathbb{R} \tag{3.2}$$

$$\mathcal{Z}(m, \neg\chi) = \mathbb{R} \setminus \mathcal{Z}(m,\chi) \tag{3.3}$$

$$\mathcal{Z}(m, \chi \wedge \omega) = \mathcal{Z}(m,\chi) \cap \mathcal{Z}(m,\omega) \tag{3.4}$$

$$\mathcal{Z}(m, \exists(\hat{n},\chi)) = \bigcup_{\hat{m}\in\hat{M}} \lambda^{\hat{m}} \cap \mathcal{Z}(\hat{m},\chi) \tag{3.5}$$

$$\mathcal{Z}(m, \chi \mathrm{U}_I \omega) = \begin{cases} \displaystyle\bigcup_{i\in\mathcal{Z}(m,\omega),\, j\in J_i} j \cap \big((j^+ \cap i) \ominus I\big) & \text{if } 0 \notin I \\ \displaystyle\bigcup_{i\in\mathcal{Z}(m,\omega)} i \cup \bigcup_{j\in J_i} j \cap \big((j^+ \cap i) \ominus I\big) & \text{if } 0 \in I \end{cases} \tag{3.6}$$

$$\mathcal{Z}(m, \chi \mathrm{S}_I \omega) = \begin{cases} \displaystyle\bigcup_{i\in\mathcal{Z}(m,\omega),\, j\in J_i} j \cap \big(({}^+j \cap i) \oplus I\big) & \text{if } 0 \notin I \\ \displaystyle\bigcup_{i\in\mathcal{Z}(m,\omega)} i \cup \bigcup_{j\in J_i} j \cap \big(({}^+j \cap i) \oplus I\big) & \text{if } 0 \in I \end{cases} \tag{3.7}$$

with:

- $\hat{m}$ a match for $\hat{n}$, and $\lambda^{\hat{m}}$ the lifespan of $\hat{m}$.
- $\hat{M}$ a set containing only matches that are *compatible* with the (enclosing) match m—see Section 2.2.2.
- J_i the set of all intervals in $\mathcal{Z}(m,\chi)$ that are either *overlapping* with or *adjacent* to some $i \in \mathcal{Z}(m,\omega)$.
- ${}^+k$ the union $\ell(k) \cup k$, i.e., making the interval k left-closed, and k^+ defined symmetrically.
- $k \oplus z = [\ell(k)+\ell(z), r(k)+r(z)]$, $k \ominus z = [\ell(k)-r(z), r(k)-\ell(z)]$ for $k, z \in \mathfrak{I}$ with $\ell(k)$ and $r(k)$ the left and right end-point of k, respectively—note that end-points are reversed in subtraction.

The intuition behind the equations for *true*, negation, and conjunction is clear. Regarding *exists*, the satisfaction span can be computed based on a subquery which searches for $\hat{n}$—the computation relies on the temporal validity of all matches $\hat{m}$ for $\hat{n}$ which are compatible with m.

For *until*, the computation is conditional on the timing constraint I. If $0 \notin I$, i.e., $\ell(I) \neq 0$, according to the semantics there is a non-empty interval for which ψ is continuously satisfied at least until ω is timely satisfied, i.e., within I. Thus, the computation includes every time point t in the intersection of some $i' \in \mathcal{Z}(m,\omega)$ with a $j' \in \mathcal{Z}(m,\chi)$ for which a time point τ' in i' occurs within I. Furthermore, j' needs to *overlap* i', e.g., $j' = [1,3], i' = [2,4]$ or be *adjacent* to i', e.g., $j' = [1,2), i' = [2,4]$. If j' and i' are adjacent, during the computation j' becomes right-closed, i.e., $j'^{+} = [1,2]$, to ensure that their intersection produces a non-empty set. If $0 \in I$, then, by the semantics, it may be that the interval for which ψ is satisfied (j') is empty, i.e., does not exist, and in that case *until* is satisfied as soon as ω is satisfied, i.e., for every $i' \in \mathcal{Z}(m,\omega)$. Therefore, the computation includes every i' and remains unchanged otherwise. The intuition behind *since* is analogous.

The following theorem states that the set of time points in the satisfaction span $\mathcal{Y}$ is equal to the set of time points obtained by the satisfaction computation $\mathcal{Z}$.

Theorem 3.2.1 (equality of satisfaction span and satisfaction computation)**.** *Given a match m in $H_{[\tau]}$ and an MTGC ψ, the satisfaction span $\mathcal{Y}$ of m for ψ over $H_{[c]}$ is given by the satisfaction computation $\mathcal{Z}$ of m for ψ over $H_{[c]}$, that is,* $\mathcal{Y}(m,\psi) = \mathcal{Z}(m,\psi)$

Proof (idea). By structural induction over ψ. For every equation, inclusion is shown in both directions. See Section A.3.1 for the complete proof. □

Thus $\mathcal{Z}$ enables the computation of all time points in $\mathbb{R}$ for which a match satisfies an MTGC, i.e., $\mathcal{Y}$. We now present a technical definition of the temporal validity $\mathcal{V}$ of a match.

Definition 3.2.2 (temporal validity $\mathcal{V}$)**.** Let n be a pattern, ψ an MTGC, and $H_{[\tau]}$ an RTM$^{\mathrm{H}}$. Moreover, let (n,ψ) be a query for a pattern n which satisfies ψ. Then, for a match m for n in $H_{[\tau]}$ with lifespan λ^m, its *temporal validity* $\mathcal{V}$, denoted as $\mathcal{V}(m,\psi)$, is an interval set defined as $\mathcal{V}(m,\psi) := \lambda^m \cap \mathcal{Z}(m,\psi)$, with $\mathcal{Z}(m,\psi)$ the satisfaction span of ψ for m.

$\mathcal{Z}(m,\psi)$ computes all time points for which m satisfies ψ. This computation may require taking the lifespan of the match m into account, but it is not bounded by it. The temporal validity $\mathcal{V}(m,\psi)$ bounds $\mathcal{Z}$ by λ^m: it computes the set of time points for which a match, *besides being structurally present* in the graph, satisfies ψ. As an example, assume the query $(n, \lozenge_{[0,5]} \exists \hat{n})$. For a match m for n with $\lambda^m = [3,9)$, the satisfaction span $\mathcal{Z}(m, \lozenge_{[0,5]} \exists \hat{n})$ for a match $\hat{m}$ for $\hat{n}$ with $\lambda^{\hat{m}} = [3,6)$ is $[-2,6)$—according to Equation 3.6. Thus, in this example, $\mathcal{Z}(m,\psi) \notin \lambda^m$. However, $\mathcal{V}(m, \lozenge_{[0,5]} \exists \hat{n})$ is equal to $\lambda^m \cap [-2,6) = [3,6)$. As the example showed, $\mathcal{Z}$ may contain negative time points (hence $\mathbb{R}$ is used in Definition 3.2.1), whereas $\mathcal{V} \subseteq \mathbb{R}_0^+$—since $\mathcal{V}$ is produced by an intersection with λ^m. It also holds that $\mathcal{V}(m,\psi) \subseteq \mathcal{Z}(m,\psi)$.

3.2.3 *Query Answer Set*

Based on the temporal validity, we can proceed with a technical definition of the output of a query in $\mathcal{L}_T$, that is, its answer set $\mathcal{T}$. In a RTM^H, all entities and, thus, all matches have a lifespan. Moreover, ACs of queries in $\mathcal{L}_T$ are formulas of an interval-based temporal logic which is capable of deciding the truth value of a formula for every time point in $\mathbb{R}$. Therefore, the answer set of a query in $\mathcal{L}_T$ contains matches associated with a temporal validity, i.e., the time points for which the match exists *and* satisfies its AC.

Definition 3.2.3 (query answer set $\mathcal{T}$)**.** Given a pattern n, an MTGC ψ, and an RTM^H $H_{[\tau]}$, the *query answer set* $\mathcal{T}$ for a query $(n, \psi) \in \mathcal{L}_T$ over $H_{[\tau]}$ is given by:

$$\mathcal{T}(H_{[\tau]}) = \{(m, \mathcal{V}(m, \psi)) | m \text{ is a match for } n \text{ in } H_{[\tau]} \wedge \mathcal{V}(m, \psi) \neq \varnothing\}$$

Recall that $\mathcal{V}(m, \psi)$ is an interval set. The answer set allows for a precise definition of the output of $\mathcal{L}_T$. In the remainder, we rely on this definition to explain the output of queries in the examples, argue on the soundness of the query evaluation, and introduce additional types of answer sets.

Example 3.2.1 (satisfaction span, temporal validity, temporal query answer set)**.** *We demonstrate the computation of the satisfaction span $\mathcal{Z}$, the temporal validity $\mathcal{V}$, and temporal query answer set $\mathcal{T}$ of $\mathcal{L}_T$ based on the query $\zeta_1 := (n_1, \psi_1)$, where $\psi_1 := \neg\exists\, n_{1.1}\, \mathrm{U}_{[0,60]}\, \exists\, n_{1.2}$, and the RTM^H $H_{[7]}$ from Figure 3.2.*

Let m_1 be a match for the query pattern n_1 involving `s` *and* `pm1` *and p_1 be a match for n_1 involving* `s` *and* `pm2`*. Moreover, let $m_{1.1}$ be a match for the pattern $n_{1.2}$ of the MTGC $\exists\, n_{1.2}$ involving* `s`*,* `pm1`*, and* `d1`*. The satisfaction span computations $\mathcal{Z}(m_1, \psi_1)$ and $\mathcal{Z}(p_1, \psi_1)$ according to Definition 3.2.1 are shown in Table 3.1. The computation for $\mathcal{Z}(m_1, \exists n_{1.2})$ is based on the intersection of the lifespan of $m_{1.2}$ with the satisfaction span $\mathcal{Z}(m_{1.2}, \mathrm{true})$. The computation for* until *is based on the overlapping intervals $\mathbb{R}$, i.e., the satisfaction computation $\mathcal{Z}(m_1, \neg\exists n_{1.1})$ for the left operand of the* until*, and $[5, 7)$, which has been computed by $\mathcal{Z}(m_1, \exists n_{1.2})$.*

The temporal validity $\mathcal{V}$ of m_1 and p_1 is computed based on their intersection of their lifespans with the satisfaction spans $\mathcal{Z}(m_1, \psi_1)$ and $\mathcal{Z}(p_1, \psi_1)$, i.e., $[4, \infty) \cap [-55, 7)$ and $[7, \infty) \cap \varnothing$, respectively. Since the answer set $\mathcal{T}$ admits only matches whose temporal validity is non-empty, the answer set $\mathcal{T}(H_{[7]})$ contains only the match m_1 paired with its temporal validity $[4, 7)$. The pair reflects that there is one match for n_1 in $H_{[7]}$ that satisfies the AC of the query, and, moreover, this match satisfies the AC for the interval $[4, 7)$.

3.3 QUERYING AN UNFINISHED HISTORY

As mentioned previously, an RTM^H provides two views on the modeled architecture: a view of the *current* state at time point τ and, by virtue of a creation and a deletion timestamp per entity, a compact view of the *history* of the state, i.e., changes to the state in the period $[0, \tau]$. The value of the deletion timestamp of entities that model components which are currently active in the system, i.e., part of the current view, is ∞. This modeling decision captures the intuition that an active component may or may not be deleted during the lifetime of a system, and if it is deleted, this will occur at a time point larger than τ, i.e., some time in the future. Technically, since the *dts* defines the lifespan of the entity, the RTM^{H} *makes an assumption about time points larger than* τ, although, in practice nothing is known about these time points.

The computation of the satisfaction span $\mathcal{Z}$ of an MTGC over the RTM^{H} instance $H_{[\tau]}$ makes a satisfaction decision for every time point in the time domain based on this assumption. However, for some of these time points, the decision may be subject to change: for time points larger than τ, because changes that will occur after τ are unknown, and for time points before τ, including τ itself, because future temporal operators may make the satisfaction check dependent on time points larger than τ. These effects of the assumption are demonstrated in the computation of the satisfaction span over $H_{[5]}$ and $H_{[7]}$: based on the lifespans of elements whose *dts* is ∞, the temporal validity $\mathcal{V}$ of the match m_1 over $H_{[5]}$ is, similarly to Example 3.2.1, computed to be $[4,\infty)$; over $H_{[7]}$, the $\mathcal{V}$ is computed to be $[4,7)$.

The results of the computation of the satisfaction span presented in Section 3.2.2 are definite, only if the RTM^{H} over which the computation is performed is the last instance of a history, i.e., it encodes a *finished* history. Given that RTMs are typically queried reactively and over a period of time whose end may be unknown, it is desirable to adjust the computation of the satisfaction span such that the result is definite also in intermediate query evaluations over an *unfinished* history. Nonetheless, we further motivate the computation presented in Section 3.2.2 later in Chapter 4, as it enables a higher degree of incrementality.

3.3.1 *Runtime Monitoring with Temporal Queries*

The objective of providing definite answers while monitoring a temporal requirement over an unfinished history demonstrates the similarity of the setting assumed in this thesis with the monitoring approach known as *Runtime Verification* (RV) [see 11]. RV typically represents the system behavior by a trace of events at some level of abstraction that are observed as they come; an online algorithm is then used to check whether the behavior satisfies a given property [13] typically specified in a temporal logic.

Table 3.1: The computation for the satisfaction span $\mathcal{Z}$ for (n_1, ψ_1) over $H_{[7]}$; m_1 and p_1 denote the matches for n_1 which involve `pm1` and `pm2`, respectively.

MTGC	m_1	p_1
true	$\mathbb{R}$	$\mathbb{R}$
$\exists n_{1.1}$	$\varnothing$	$\varnothing$
$\neg\exists n_{1.1}$	$\mathbb{R}$	$\mathbb{R}$
true	$\mathbb{R}$	$\mathbb{R}$
$\exists n_{1.2}$	$[5,7)$	$\varnothing$
$\neg\exists_{1.1} \mathrm{U}_{[0,60]} \exists_{1.2}$	$[-55,7)$	$\varnothing$

In our context, this trace corresponds to the history. As the behavior may be incomplete, i.e., the trace encodes an unfinished history, a major issue in RV is the extension of temporal logic semantics, which are defined over complete and infinite traces, to incomplete behaviors [106]. Moreover, RV requires that the given property is *monitorable*, i.e., a definite satisfaction or definite falsification can be detected after observing a finite trace; satisfaction is detected when any possible extension of the observed trace satisfies the property, whereas falsification is detected when the observed trace cannot be extended in any way such that it satisfies the property [see 122]. A class of such properties is *safety properties* [2], i.e., statements of the form "something will not happen" [100], where falsification can be detected as soon as it occurs [99].

In the context of graphs, a safety property with no temporal operators corresponds to a graph query in $\mathcal{L}$, i.e., a purely structural query—see Section 2.2.2. Monitoring such properties corresponds to updating a host graph representing the system state based on a sequence of events and, after each update, searching the graph for matches of the query pattern [33]. If a match is found, then the property is falsified, whereas the absence of matches implies a non-definite satisfaction, as the property could be falsified in the future.

Monitoring properties with temporal operators require the support for the specification and evaluation of temporal queries, i.e., queries that combine structure and requirements on the evolution of structure over time, such as those specified in $\mathcal{L}_T$. Similarly to [33], monitoring would require the evaluation of a temporal query following each change to the host graph, which is the objective of the querying approach presented in the next chapter.

Moreover, for the considered use-cases, a richer semantics that goes beyond a two-valued interpretation based on *false* or *true* and captures the possibility of a decision being non-definite would be useful. For this reason, it is often the case that RV algorithms support a three-valued satisfaction check which returns *true* when the property is satisfied, *false* when it is falsified, and *unknown* when the result of the satisfaction check is not definite, i.e., when the observed trace could be extended to satisfy the property but also to falsify it [16, 54]. In the following, we introduce a three-valued semantics for $\mathcal{L}_T$ and adjust the computation of the satisfaction span such that it is suitable for monitoring temporal properties over unfinished histories.

3.3.2 *Definite Satisfaction and Definite Falsification*

Similarly to the three-valued satisfaction check in RV, we equip query evaluations with a capability to compute the *falsification span*, i.e., the time points for which a property is falsified. Moreover, to adjust the query evaluation to unfinished histories, we introduce the *definite satisfaction span* and the *definite falsification span* which contain only those time points for which the satisfaction or falsification, respectively, is definite, i.e., any possible continuation of the history will have no impact on the satisfaction decision. In the context of an RTM$^{\mathrm{H}}$ $H_{[c]}$, a satisfaction decision at time point τ is *definite* if the decision at τ remains the same in all possible future versions of $H_{[c]}$.

We obtain the definite satisfaction span by adjusting the satisfaction relation of MTGL from Definition 2.3.3 to this notion of definiteness. Moreover, we obtain the definite falsification span by negating the statements in the cases of the definite satisfaction. We first present the adjusted satisfaction relation, called *definite satisfaction relation*, and the *definite falsification relation* over an RTM$^{\mathrm{H}}$, i.e., a graph with history, below.

Definition 3.3.1 (definite satisfaction and definite falsification of metric temporal graph conditions over an RTM$^{\mathrm{H}}$)**.** Let $H_{[c]}$ be an RTM$^{\mathrm{H}}$, n a pattern, and $m : n \hookrightarrow H_{[c]}$ a match. Moreover, let τ be a time point in $\mathbb{R}$ and ψ be an MTGC over n. Then m *definitely satisfies* ψ at τ, written $(m, \tau) \models^d \psi$, if and only if $\tau \in \lambda^m \cap [0, c]$, or m is the empty match, and one of the following cases applies.

- $\psi = \mathit{true}$.
- $\psi = \neg\chi$ and $(m, \tau) \models^d_F \chi$.
- $\psi = \chi \wedge \omega$, $(m, \tau) \models^d \chi$, and $(m, \tau) \models^d \omega$.
- $\psi = \exists(f : n \hookrightarrow \hat{n}, \chi)$ and there exists $\hat{m} : \hat{n} \hookrightarrow H_{[c]}$ such that $\hat{m} \circ f = m$ and $(\hat{m}, \tau) \models^d \chi$.
- $\psi = \chi \mathrm{U}_I \omega$ and there exists τ' with $\tau' - \tau \in I$ such that $(m, \tau') \models^d \omega$ and for all $\tau'' \in [\tau, \tau')$ $(m, \tau'') \models^d \chi$.
- $\psi = \chi \mathrm{S}_I \omega$ and there exists τ' with $\tau - \tau' \in I$ such that $(m, \tau') \models^d \omega$ and for all $\tau'' \in (\tau', \tau]$ $(m, \tau'') \models^d \chi$.

In comparison to the semantics of MTGL presented in Definition 2.3.3, the definite satisfaction semantics confine the lifespans of matches and the satisfaction of *exists* to the period that has been observed, i.e., $[0, c]$. Moreover, the definite satisfaction relation relies on the definite falsification relation for the satisfaction of a negation.

The definite falsification relation is based on a logical negation of the statements in the cases of the definite satisfaction relation. The match m *definitely falsifies* ψ at τ, written $(m, \tau) \models^d_F \psi$, if and only if $\tau \in \lambda^m \cap [0, c]$, or m is the empty match, and one of the following cases applies.

- $\psi = \neg\chi$ and $(m, \tau) \models^d \chi$.
- $\psi = \chi \wedge \omega$ and $(m, \tau) \models^d_F \chi$ or $(m, \tau) \models^d_F \omega$.
- $\psi = \exists(f : n \hookrightarrow \hat{n}, \chi)$ and either there does not exist an $\hat{m} : \hat{n} \hookrightarrow H_{[c]}$ such that $\hat{m} \circ f = m$, or there exists $\hat{m}$ and $(\hat{m}, \tau) \models^d_F \chi$.
- $\psi = \chi \mathrm{U}_I \omega$ and for all τ' with $\tau' - \tau \in I$ $(m, \tau') \models^d_F \omega$ or there exists $\tau'' \in [\tau, \tau')$ such that $(m, \tau'') \models^d_F \chi$.
- $\psi = \chi \mathrm{S}_I \omega$ and for all τ' with $\tau - \tau' \in I$ $(m, \tau') \models^d_F \omega$ or there exists $\tau'' \in (\tau', \tau]$, $(m, \tau'') \models^d_F \chi$.

Intuitively, the match m never falsifies *true*. Moreover, similarly to the definite satisfaction relation $\models^d$, the definite falsification relation confines the decisions that concern matches to $[0, c]$, and relies on $\models^d$ for the falsification of negation.

The definite falsification relation ($\models^d_F$) and the negation of the definite satisfaction relation ($\not\models^d$) are not equivalent; $\not\models^d$ returns true for time points that do not definitely satisfy the operator, i.e., points that falsify it but also points for which a definite decision cannot yet be made.

The following results relate the definite satisfaction relation $\models^d$ and definite falsification relation $\models^d_F$, i.e., the *definite relations*, to the satisfaction relation $\models$ and its negation $\not\models$. The results refer to observed prefixes of a possibly infinite sequence of RTM$^{\mathrm{H}}$ instances h^H and their possible continuations; an RTM$^{\mathrm{H}}$ instance $H_{[\tau_i]}$ in h^H is associated with the timestamp of the event with index $i \in \mathbb{N}^+$ in the history—see Section 2.1.3 and Section 3.1.

The following theorem states that a *definite decision*, i.e., a decision made by either the definite satisfaction or the definite falsification relation, for a certain time point τ over an $H_{[\tau_i]}$ in h^H implies that the same decision is made by the satisfaction relation (or its negation) for τ over $H_{[\tau_i]}$; moreover, the satisfaction relation makes the same decision for τ over all possible future versions of $H_{[\tau_i]}$ in h^H. In the following, we extend the syntax of relations for clarity and write, e.g., $(H_{[\tau_i]}, m, \tau) \models \psi$ instead of $(m, \tau) \models \psi$.

Theorem 3.3.1 (definite relations imply satisfaction relation over history)**.** *Let ψ be an MTGC over a pattern n. Moreover, let $h^H_{\tau_{\mathcal{D}}}$ be a sequence of RTM$^{\mathrm{H}}$ instances, with $\mathcal{D} \in \mathbb{N}^+$. For all $i \in [1, \mathcal{D}] \cap \mathbb{N}^+$, if m is a match for n in $H_{[\tau_i]}$ and $\tau \in [0, \tau_i]$, then for all $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$, (i) if $(H_{[\tau_i]}, m, \tau) \models^d \psi$, then $(H_{[\tau_k]}, m, \tau) \models \psi$, and (ii) if $(H_{[\tau_i]}, m, \tau) \models^d_F \psi$, then $(H_{[\tau_k]}, m, \tau) \not\models \psi$.*

Proof (idea). By mutual structural induction over ψ. The implication is shown to hold for each MTGL operator. See Section A.3.2 for the complete proof. □

As mentioned previously, the satisfaction decision for future temporal operators at time point τ may depend on a $\tau' > \tau$. The upper bound of the distance between τ' and τ is given by the *non-definiteness window*, defined below.

Definition 3.3.2 (non-definiteness window w)**.** Given an MTGC ψ, the *non-definiteness window* w, i.e., the period for which a satisfaction decision for ψ at a time point τ may be non-definite, is defined as follows.

$$w(\psi) = \begin{cases} r(I) + \max(w(\chi), w(\omega)) & \text{if } \psi = \chi \mathrm{U}_I \, \omega \\ \max(w(\chi), w(\omega)) & \text{if } \psi = \chi \mathrm{S}_I \, \omega \\ \max(w(\chi), w(\omega)) & \text{if } \psi = \chi \wedge \omega \\ w(\chi) & \text{if } \psi = \neg \chi \\ w(\chi) & \text{if } \psi = \exists(n, \chi) \\ 0 & \text{if } \psi = \mathit{true} \end{cases} \tag{3.8}$$

As is typically the case in (online) runtime monitoring, we assume that $w \neq \infty$, i.e., MTGCs contain no unbounded future operators, which may render a property non-monitorable—see Section 3.3.1.

Based on the non-definiteness time window, we present a variation of Theorem 3.3.1 which states that, given an $H_{[\tau_i]}$, if $\tau \in [0, \tau_i - w]$, with i an index in a sequence of RTM$^{\mathrm{H}}$ instances, then definite decisions made by either the definite satisfaction or definite falsification relation are equivalent to the decisions of the satisfaction relation. If $w \neq 0$, in order for $[0, \tau_i - w]$ to be a valid interval, it is implicitly required that $\tau_i \geq w$, i.e., $H_{[\tau_i]}$ covers a period that is larger than the non-definiteness window.

Theorem 3.3.2 (definite relations are equivalent to satisfaction relation over certain period of history)**.** *Let ψ be an MTGC over a pattern n and w the non-definiteness window of ψ. Moreover, let $h^H_{\tau_\mathcal{D}}$ be a sequence of RTM^H instances, with $\mathcal{D} \in \mathbb{N}^+$. For all $i \in [1, \mathcal{D}] \cap \mathbb{N}^+$, if m is a match for n in $H_{[\tau_i]}$ and $\tau \in [0, \tau_i - w]$, then for all $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$, (i) $(H_{[\tau_i]}, m, \tau) \models^d \psi$ if and only if $(H_{[\tau_k]}, m, \tau) \models \psi$, and (ii) $(H_{[\tau_i]}, m, \tau) \models^d_F \psi$ if and only if $(H_{[\tau_k]}, m, \tau) \not\models \psi$.*

Proof (idea). By mutual structural induction over ψ. The equivalence is shown to hold for each MTGL operator. See Section A.3.3 for the complete proof. □

Theorem 3.3.2 allows for the application of techniques which increase incrementality in the evaluation of queries, and is motivated further in Section 4.2.3.

The following corollary states that all time points for which a definite decision cannot be made belong to a certain period within the observed history.

Corollary 3.3.1 (period in history with non-definite decisions)**.** *Let ψ be an MTGC, w be the non-definiteness window of ψ, $H_{[\tau_i]}$ be an RTM^H instance associated with the time point τ_i, m be a match for a pattern n, and τ a time point in $[0, \tau_i]$. If $(H_{[\tau_i]}, m, \tau) \not\models^d \psi$ and $(H_{[\tau_i]}, m, \tau) \not\models^d_F \psi$, then $\tau \in (\tau_i - w, \tau_i]$.*

Proof. Follows from Theorem 3.3.2. The satisfaction relation and its negation make a decision for every time point in $[0, \tau_i - w]$, i.e., the relation does not support the value *unknown*—see Section 3.3.1; Theorem 3.3.2 shows that the decisions made by the satisfaction relation and its negation for $[0, \tau_i - w]$ are equivalent to the decisions made by the definite relations. Consequently, if no definite decision is made for $\tau \in [0, \tau_i]$, then $\tau \notin [0, \tau_i - w]$. □

Let $\models_\mathbb{T}$ and $\models_{F,\mathbb{T}}$ be respectively a satisfaction and falsification relation for MTGL that reflect the timeliest knowledge: Given a match m, an MTGC ψ, an RTM$^\text{H}$ instance $H_{[\tau_i]}$ from a sequence of instances, and a time point $\tau \in [0, \tau_i]$, $(H_{[\tau_i]}, m, \tau) \models_\mathbb{T} \psi$ if $(H_{[\tau_i]}, m, \tau) \models \psi$ and there exists no possible successor of $H_{[\tau_i]}$ in the sequence that could falsify ψ at τ; analogously, $(H_{[\tau_i]}, m, \tau) \models_{F,\mathbb{T}} \psi$ if $(H_{[\tau_i]}, m, \tau) \not\models \psi$ and there exists no possible successor of $H_{[\tau_i]}$ that could satisfy ψ at τ. These timeliest relations can only make decisions for m over the observed history, as m may not exist in the parts covered by successors of $H_{[\tau_i]}$, i.e., in time points larger than τ_i.

Given a sequence of RTM$^\text{H}$ instances h^H with $H_{[\tau_i]}$ an instance in h^H, let $H_{[\tau_k]}$ be the first successor of $H_{[\tau_i]}$ in h^H for which $\tau_k \geq \tau_i + w$. The following corollary states that, contrary to $\models_\mathbb{T}$ and $\models_{F,\mathbb{T}}$, the definite relations may have to wait for $H_{[\tau_k]}$ to be able to make a definite decision for $\tau \in (\tau_i - w, \tau_i]$.

Corollary 3.3.2 (maximum possible wait before definite decision)**.** *Let ψ be an MTGC, w be the non-definiteness window of ψ, m be a match for a pattern n, and $H_{[\tau_i]}$ be an RTM^H instance from a sequence of RTM^H instances $h^H_{\tau_\mathcal{D}}$ with $i \in [1, \mathcal{D}] \cap \mathbb{N}^+$. Moreover, let $\tau \in (\tau_i - w, \tau_i]$ and k be the smallest index in $[i, \mathcal{D}] \cap \mathbb{N}^+$ such that $\tau_k \geq \tau_i + w$. If $(H_{[\tau_i]}, m, \tau) \not\models^d \psi$ and $(H_{[\tau_i]}, m, \tau) \not\models^d_F \psi$, then a definite decision for τ can be made over $H_{[\tau_k]}$.*

Proof. Follows from Corollary 3.3.1. □

Thus, compared to $\models_{\mathbb{T}}$ and $\models_{F,\mathbb{T}}$, the definite relations may make a decision for $\tau \in (\tau_i - w, \tau_i]$ with a delay of at most $(\tau_k - \tau_i)$ time points.

Example 3.3.1. *(delay in definite decision) Let $\psi_c := \lozenge_{[0,1]}(\neg\exists n_1 \wedge \exists n_1)$. Consider a sequence consisting of two RTM^H instances: $H_{[7]}$ in Figure 3.2 and a hypothetical $H_{[9]}$ which is yielded by an unrelated change and all elements from $H_{[7]}$ are unchanged. Therefore, a match m_1 exists in both instances. The check $(H_{[7]}, m_1, 7) \models_{F,\mathbb{T}} \psi_c$ returns true, as $(H_{[7]}, m_1, 7) \not\models \psi_c$ and there is no possible successor of $H_{[7]}$ that could satisfy ψ_c; on the other hand, $(H_{[7]}, m_1, 7) \models^d_F \psi_c$ makes no decision, as according to its definition, the relation waits first for a duration of history that covers the timing constraint of* until *to be observed. The check $(H_{[9]}, m_1, 7) \models^d_F \psi_c$ returns true, as enough time has elapsed. Thus, compared to $\models_{F,\mathbb{T}}$, this decision has been made with a delay of two time points.*

The delay is observed with MTGCs which are unsatisfiable or unfalsifiable, e.g., ψ_c from Example 3.3.1 which contains a contradiction. Avoiding this delay would require that the definite relations recognize whether an MTGC is satisfiable which is undecidable for NGCs and thus MTGCs. The delay is not observed with the running example, i.e., $\psi_1 := \neg\exists n_{1.1} \, \mathrm{U}_{[0,60]} \exists n_{1.2}$ or similar MTGCs, as demonstrated by the following example.

Example 3.3.2. *(timely definite decision) Let $\psi_{nc} := (\lozenge_{[0,2]}\exists n_{1.1}) \wedge (\lozenge_{[0,3]}\exists n_{1.2})$ over a context pattern n_1. Consider the same sequence of two RTM^H instances as in Example 3.3.1, i.e., comprising $H_{[7]}$ and $H_{[9]}$. Let m_1 be a match for n_1 in $H_{[7]}$; $H_{[7]}$ and $H_{[9]}$ contain no matches for $n_{1.1}$ or $n_{1.2}$. The check $(H_{[7]}, m_1, 7) \models_{F,\mathbb{T}} \psi_{nc}$ does not detect a falsification as there is a possible successor of $H_{[7]}$ that could satisfy ψ_{nc} at time point 7; check $(H_{[9]}, m_1, 7) \models_{F,\mathbb{T}} \psi_{nc}$ returns true as $(H_{[9]}, m_1, 7) \not\models \psi_{nc}$ and there is no possible successor of $H_{[9]}$ that could satisfy the left operand of the conjunction at time point 7. Similarly, $(H_{[7]}, m_1, 7) \models^d_F \psi_{nc}$ does not return true and $(H_{[9]}, m_1, 7) \models^d_F \psi_{nc}$ does. Therefore, $\models^d_F$ makes the decision timely, i.e., at the same time point as $\models_{F,\mathbb{T}}$.*

The delay could also be avoided by a reformulation of a ψ with $w \neq 0$ into a ψ' with $w = 0$, i.e., without any future temporal operators. However, although technically possible, such a reformulation may be non-trivial in metric temporal logics [see 4, 90] and especially in the presence of bindings.

3.3.3 *Definite Satisfaction Span and Definite Falsification Span*

In the context of a query $(n, \psi) \in \mathcal{L}_{\mathrm{T}}$ or a nested condition with n as enclosing pattern, the *definite satisfaction span* related to a match m for n is defined similarly to $\mathcal{Y}$ in Section 3.2.2, i.e., $\mathcal{Y}^d = \{\tau | \tau \in \mathbb{R} \wedge (m, \tau) \models^d \psi\}$. The *definite falsification span* is defined as $\mathcal{F} = \{\tau | \tau \in \mathbb{R} \wedge (m, \tau) \models^d_F \psi\}$. Any time point in the time domain not in $\mathcal{Y}^d$ or $\mathcal{F}$ belongs to the *unknown span* X. The sets $\mathcal{Y}^d, \mathcal{F}$, and X are disjoint. It also holds that $\mathbb{R} = \mathcal{Y}^d \uplus \mathcal{F} \uplus X$.

Similarly to the satisfaction span $\mathcal{Y}$ and the satisfaction computation $\mathcal{Z}$ in Section 3.2.2, we present the *definite satisfaction computation* $\mathcal{Z}^d$ and the *definite falsification computation* F for an MTGC.

Definition 3.3.3 (definite satisfaction computation $\mathcal{Z}^d$ and definite falsification computation F)**.** Let $n, \hat{n}$ be patterns and ψ, χ, ω be MTGCs. Moreover, let $H_{[\tau]}$ be an RTM$^{\mathrm{H}}$, m be a match for n in $H_{[\tau]}$, and $F(m,\psi)$ the definite falsification computation of m for ψ. The *definite satisfaction computation* $\mathcal{Z}^d(m,\psi)$ is defined as follows.

$$\mathcal{Z}^d(m, true) = \mathbb{R} \tag{3.9}$$

$$\mathcal{Z}^d(m, \neg\chi) = F(m,\chi) \tag{3.10}$$

$$\mathcal{Z}^d(m, \chi \wedge \omega) = \mathcal{Z}^d(m,\chi) \cap \mathcal{Z}^d(m,\omega) \tag{3.11}$$

$$\mathcal{Z}^d(m, \exists(\hat{n},\chi)) = (-\infty, \tau] \cap \bigcup_{\hat{m} \in \hat{M}} \lambda^{\hat{m}} \cap \mathcal{Z}^d(\hat{m},\chi) \tag{3.12}$$

$$\mathcal{Z}^d(m, \chi \mathrm{U}_I \omega) = \begin{cases} \bigcup\limits_{i \in \mathcal{Z}^d(m,\omega),\, j \in J_i^d} j \cap \big((j^+ \cap i) \ominus I\big) & \text{if } 0 \notin I \\ \bigcup\limits_{i \in \mathcal{Z}^d(m,\omega)} i \cup \bigcup\limits_{j \in J_i^d} j \cap \big((j^+ \cap i) \ominus I\big) & \text{if } 0 \in I \end{cases} \tag{3.13}$$

$$\mathcal{Z}^d(m, \chi \mathrm{S}_I \omega) = \begin{cases} \bigcup\limits_{i \in \mathcal{Z}^d(m,\omega),\, j \in J_i^d} j \cap \big(({}^+j \cap i) \oplus I\big) & \text{if } 0 \notin I \\ \bigcup\limits_{i \in \mathcal{Z}^d(m,\omega)} i \cup \bigcup\limits_{j \in J_i^d} j \cap \big(({}^+j \cap i) \oplus I\big) & \text{if } 0 \in I \end{cases} \tag{3.14}$$

where, similarly to Definition 3.2.1, $\hat{m}$ is a match for $\hat{n}$, and $\lambda^{\hat{m}}$ the lifespan of $\hat{m}$, $\hat{M}$ a set containing only matches that are compatible with the (enclosing) match m—see Section 2.2.1, and J_i^d the set of all intervals in $\mathcal{Z}^d(m,\chi)$ that are either *overlapping* with or *adjacent* to some $i \in \mathcal{Z}^d(m,\omega)$.

The equations for conjunction, *until*, and *since* have the same structure with their corresponding equations in Definition 3.2.1, but rely on $\mathcal{Z}^d$ instead of $\mathcal{Z}$. Analogously to the definite satisfaction relation, the computation for negation relies on the definite falsification computation. The computation for *exists* confines its decisions to the period that has been observed.

Based on $\mathbb{R} = \mathcal{Y}^d \uplus \mathcal{F} \uplus X$, the *definite falsification computation* $F(m, \psi)$ can be generally defined as $F = \mathbb{R} \setminus (\mathcal{Z}^d \uplus X)$, which leads to the following equations.

$$F(m, true) = \varnothing \tag{3.15}$$

$$F(m, \neg\chi) = \mathcal{Z}^d(m, \chi) \tag{3.16}$$

$$F(m, \chi \wedge \omega) = F(m, \chi) \cup F(m, \omega) \tag{3.17}$$

$$F(m, \exists(\hat{n}, \chi)) = (-\infty, \tau] \cap \left(\mathbb{R} \setminus \mathcal{Z}^d(m, \exists(\hat{n}, \chi))\right) \tag{3.18}$$

$$F(m, \chi \mathrm{U}_I \omega) = \begin{cases} \mathbb{R} \setminus \left(\bigcup\limits_{i \in \mathcal{Z}^d(m,\omega) \uplus X(m,\omega),\, j \in J_i^d} j \cap \left((j^+ \cap i) \ominus I \right) \right) & \text{if } 0 \notin I \\ \mathbb{R} \setminus \left(\bigcup\limits_{i \in \mathcal{Z}^d(m,\omega) \uplus X(m,\omega)} i \cup \bigcup\limits_{j \in J_i^d} j \cap \left((j^+ \cap i) \ominus I \right) \right) & \text{if } 0 \in I \end{cases} \tag{3.19}$$

$$F(m, \chi \mathrm{S}_I \omega) = \begin{cases} \mathbb{R} \setminus \left(\bigcup\limits_{i \in \mathcal{Z}^d(m,\omega) \uplus X(m,\omega),\, j \in J_i^d} j \cap \left(({}^+j \cap i) \oplus I \right) \right) & \text{if } 0 \notin I \\ \mathbb{R} \setminus \left(\bigcup\limits_{i \in \mathcal{Z}^d(m,\omega) \uplus X(m,\omega)} i \cup \bigcup\limits_{j \in J_i^d} j \cap \left(({}^+j \cap i) \oplus I \right) \right) & \text{if } 0 \in I \end{cases} \tag{3.20}$$

where J_i^d is the set of all intervals in $\mathcal{Z}^d(m, \chi) \uplus X(m, \chi)$ that are either *overlapping* with or *adjacent* to some $i \in \mathcal{Z}^d(m, \omega) \uplus X(m, \omega)$.

A match m never falsifies *true*; analogously to the definite falsification relation, the definite falsification computation relies on the definite satisfaction computation for the falsification of negation; the operator *exists* confines its computation to the observed period; the equations for *until* and *since* complement the definite satisfaction computation for the operators, whereby the definite satisfaction computation for their operands χ and ω, instead of considering only time points that definitely satisfy χ and ω, i.e., their satisfaction span $\mathcal{Z}^d(m, \chi)$ and $\mathcal{Z}^d(m, \omega)$, considers time points *that do not definitely falsify* χ and ω, i.e., $\mathcal{Z}^d(m, \chi) \uplus X(m, \chi)$ and $\mathcal{Z}^d(m, \omega) \uplus X(m, \omega)$.

The following theorem states that the set of time points in the definite satisfaction span $\mathcal{Y}^d$ and definite falsification span $\mathcal{F}$ are equal to the sets of time points obtained by the definite satisfaction computation $\mathcal{Z}^d$ and definite falsification computation F, respectively.

Theorem 3.3.3 (equality of definite spans and definite computations for satisfaction and falsification)**.** *Given a match m in an RTM^H $H_{[\tau]}$ and an MTGC ψ, the definite satisfaction span $\mathcal{Y}^d$ of m for ψ over $H_{[\tau]}$ is given by the definite satisfaction computation $\mathcal{Z}^d$ of m for ψ over $H_{[\tau]}$, that is, $\mathcal{Y}^d(m, \psi) = \mathcal{Z}^d(m, \psi)$.*

Moreover, the definite falsification span $\mathcal{F}$ of m for ψ over $H_{[\tau]}$ is given by the definite falsification computation F of m for ψ over $H_{[\tau]}$, that is, $\mathcal{F}(m, \psi) = F(m, \psi)$.

Proof (idea). The proof for the definite satisfaction $\mathcal{Z}^d$ proceeds by structural induction over ψ, similarly to Theorem 3.2.1. The proof for the definite falsification F is based on the application of $F = \mathbb{R} \setminus (\mathcal{Z}^d \uplus X)$ for each MTGL operator.

The unknown span X is known for *true*, where $X = \varnothing$, and *exists*, where, by definition of the RTM$^{\mathrm{H}}$ $H_{[\tau]}$, $X = (\tau, \infty)$. If F is known, it can be used to compute $\mathcal{Z}^d \uplus X$. See Section A.3.4 for the complete proof. $\square$

3.3.4 *Temporal Invalidity and Definite Answer Set*

We define the notion of *temporal invalidity* $\mathcal{TV}$ as the dual notion of temporal validity from Definition 3.2.2, i.e., the intersection of the lifespan of a match with the definite falsification span. Based on the definite temporal validity $\mathcal{V}^d$, i.e., the intersection of the lifespan λ^m of a match m with the definite satisfaction computation $\mathcal{Z}^d$, and the *definite temporal invalidity* $\mathcal{TV}^d$, i.e., the intersection of λ^m with the definite falsification computation F, the *definite answer set* $\mathcal{T}^d$ for $\mathcal{L}_{\mathrm{T}}$ pairs matches with both their $\mathcal{V}^d$ and $\mathcal{TV}^d$. A match is admitted to $\mathcal{T}^d$ if either its $\mathcal{V}^d$ or its $\mathcal{TV}^d$ is not empty.

Definition 3.3.4 (definite answer set $\mathcal{T}^d$)**.** Given a pattern n, an MTGC ψ, and an RTM$^{\mathrm{H}}$ $H_{[\tau]}$, the *definite answer set* $\mathcal{T}^d$ for a query $(n, \psi) \in \mathcal{L}_{\mathrm{T}}$ over $H_{[\tau]}$ is the set of all triples $(m, \mathcal{V}^d(m,\psi), \mathcal{TV}^d(m,\psi))$ such that (i) m is a match for n and (ii) $(\mathcal{V}^d(m,\psi) \neq \varnothing) \vee (\mathcal{TV}^d(m,\psi) \neq \varnothing)$.

Example 3.3.3 (definite computations and definite answer set)**.** *We demonstrate the definite satisfaction computation $\mathcal{Z}^d$ and the definite falsification computation F. In this example, the query $\zeta_1 := (n_1, \psi_1)$ is evaluated over the RTM^H $H_{[5]}$ from Figure 3.2. As with Example 3.2.1, let m_1 be the match for the query pattern n_1 involving* `pm1`*, and $m_{1.1}$ be the match for $n_{1.2}$ involving* `d1`*.*

The computations for $\mathcal{Z}(m_1,\psi_1)$, $\mathcal{Z}^d(m_1,\psi_1)$, $F(m_1,\psi_1)$ are shown in Table 3.2. An important difference between the computations for $\mathcal{Z}(m_1, \exists n_{1.1})$ and $\mathcal{Z}^d(m_1, \exists n_{1.1})$ is that the definite computation removes any assumptions made by $\mathcal{Z}(m_1, \exists n_{1.1})$ with respect to the future of m_1; this computation also affects the computation for until*, and, in turn, the temporal validity. The temporal validity $\mathcal{V}$ of m_1 is $[4, \infty]$, whereas the definite temporal validity $\mathcal{V}^d$ is $[4, 5]$. The temporal invalidity $\mathcal{TV}^d$ is empty. The definite answer set $\mathcal{T}^d$ contains the match paired*

Table 3.2: The computation for the definite satisfaction computation $\mathcal{Z}^d$, definite falsification computation F, and (for reference) the satisfaction computation $\mathcal{Z}$ for (n_1, ψ_1) over $H_{[5]}$; all results concern the match m_1 for n_1 which includes the entity `pm1`.

MTGC	$\mathcal{Z}$	$\mathcal{Z}^d$	F
true	$\mathbb{R}$	$\mathbb{R}$	$\varnothing$
$\exists n_{1.1}$	$\varnothing$	$\varnothing$	$(-\infty, 5]$
$\neg\exists n_{1.1}$	$\mathbb{R}$	$(-\infty, 5]$	$\varnothing$
true	$\mathbb{R}$	$\mathbb{R}$	$\varnothing$
$\exists n_{1.2}$	$[5, \infty)$	$[5, 5]$	$(-\infty, 5)$
$\neg\exists_{1.1}\mathrm{U}_{[0,60]}\exists_{1.2}$	$[-55, \infty)$	$[-55, 5]$	$(-\infty, -55)$

with its definite temporal validity and definite temporal invalidity, i.e., the triple $(m_1, [4,5], \varnothing)$.

3.4 SUMMARY

In Section 3.1, we presented the RTM$^{\mathrm{H}}$, i.e., a compact encoding of the history of an architectural RTM (contribution C1 in Section 1.3). Entities in an RTM$^{\mathrm{H}}$ have a creation (*cts*) and a deletion (*dts*) timestamp. The lifespan of an entity e is the interval $[e.cts, e.dts)$. When an entity is created, its *dts* is assigned the value ∞; this value is updated if the component of the architecture corresponding to the entity is removed, i.e., entities are not deleted by default in an RTM$^{\mathrm{H}}$. Thus, the RTM$^{\mathrm{H}}$ affords a compact encoding of past versions of the state of the architecture.

In Section 3.2, we introduced the query language $\mathcal{L}_{\mathrm{T}}$ (C2.1), which allows for the specification of queries with temporal requirements over an RTM$^{\mathrm{H}}$. The answer for a query contains matches which satisfy the temporal requirements paired with an interval that contains time points for which a match exists and satisfies the requirements. The syntax of a query in $\mathcal{L}_{\mathrm{T}}$ is (n, ψ) where n is a query pattern, i.e., the pattern for which the query searches, and ψ is an MTGC, i.e., a temporal formula in MTGL which captures the temporal requirements on the evolution of n. Section 3.2 introduces concepts that are key to the rest of the thesis, hence we summarize them briefly and, when applicable, explain their relationship to graph- or logic-based concepts.

- *Match* (denoted by m in examples): A graph morphism tracking an occurrence of a pattern in a host graph, here the RTM$^{\mathrm{H}}$.
- *Lifespan of a match* (λ^m): The intersection of the lifespans of all entities in a match.
- *Satisfaction span* ($\mathcal{Z}(m, \psi)$): The set of time points for which a match m in a given RTM$^{\mathrm{H}}$ satisfy ψ according to the satisfaction relation of MTGL, i.e., $\{\tau | \tau \in \mathbb{R} \wedge (m, \tau) \models \psi\}$.
- *Temporal validity* ($\mathcal{V}(m, \psi)$): The intersection of the lifespan of the match λ^m with the satisfaction span $\mathcal{Z}(m, \psi)$.
- *Answer set in* $\mathcal{L}_{\mathrm{T}}$ ($\mathcal{T}$): Given the query (n, ψ), the answer set in $\mathcal{L}_{\mathrm{T}}$ contains matches for n paired with their temporal validity, provided their temporal validity is not empty.

Theorem 3.2.1 shows that the satisfaction computation we propose in Definition 3.2.1 correctly computes the satisfaction span, thereby rendering the computation of temporal validity (C2.2) sound.

In Section 3.3, we extend the $\mathcal{L}_{\mathrm{T}}$ to cover the typical case where an RTM$^{\mathrm{H}}$ is queried after every change to the model. In that case, the answer set has to reflect that future changes to the model may affect answers. To this end, we extend MTGL by the definite satisfaction relation ($\models^d$) and the definite falsification relation ($\models^d_F$) which enable a three-valued interpretation of the logic (C2.3). We summarize the introduced concepts below.

- *Definite satisfaction*: A match m in a given RTM$^{\mathrm{H}}$ definitely satisfies an MTGC ψ at τ, i.e., $(m, \tau) \models^d \psi$ if m will satisfy ψ at τ in all future versions of the RTM$^{\mathrm{H}}$.
- *Definite falsification*: A match m in a given RTM$^{\mathrm{H}}$ definitely falsifies an MTGC ψ at τ, i.e., $(m, \tau) \models^d_F \psi$ if m will falsify ψ at τ in all future versions of the RTM$^{\mathrm{H}}$.
- *Non-definiteness window* (w): For future temporal operators in MTGL, the satisfaction at time point τ may depend on a future time point τ'. The non-definiteness window is the upper bound of the distance between τ and τ' for a given MTGC.

The definite relations are interdependent. Theorem 3.3.1 shows that definite satisfaction ($\models^d$) at a time point τ over an RTM$^{\mathrm{H}}$ in a given sequence implies satisfaction ($\models$) at τ for all future RTM$^{\mathrm{H}}$ instances in the sequence; analogously, definite falsification implies negation of satisfaction ($\not\models$). Moreover, Theorem 3.3.2 shows that, if τ is from a specific time period derived based on the non-definiteness window, definite satisfaction at τ over an RTM$^{\mathrm{H}}$ in a given sequence is equivalent to satisfaction at τ for all future RTM$^{\mathrm{H}}$ instances in the sequence; analogously, definite falsification is equivalent to negation of satisfaction. Later in the thesis, it is demonstrated that this last result enables more efficient query evaluations.

Analogously to Section 3.2, in Section 3.3 we introduce the *definite satisfaction span* ($\mathcal{Z}^d$) and the *definite falsification span* (F); we define their computations and show in Theorem 3.3.3 that these computations are sound. Based on $\mathcal{Z}^d$ and F, we introduce the *definite temporal validity* and *definite temporal invalidity*, respectively. Given the query (n, ψ), the *definite answer set* ($\mathfrak{T}^d$) (C2.4 in Section 1.3) contains matches for n paired with their definite temporal validity and invalidity, provided that one of them is not empty.

4 QUERYING APPROACH

This chapter presents a querying approach for temporal queries in $\mathcal{L}_T$. The approach is a sequence of inter-dependent operations which perform the query evaluation over an RTM^H, and called InTempo from *Incremental evaluation of TEMPoral model queries*. In the following, we present an overview of InTempo—see Figure 4.1 for a graphical reference.

InTempo assumes the following artifacts have been made available at *design-time* (i) a *metamodel* of the system with creation and deletion timestamps for each entity (ii) a *mapping* of events capturing system changes to model changes, i.e., additions or deletions of entities and connectors.

The approach represents the history of the system by an RTM^H, over which it evaluates a given set of input queries in $\mathcal{L}_T$. The operation of InTempo at runtime is *reactive*. Upon the occurrence of an event, InTempo consults the event mapping, makes the corresponding structural changes to the RTM^H, and sets the values of creation and deletion timestamps of the modified entities based on the time point of the event. Subsequently, InTempo evaluates the input queries for each change and returns the answers to the system.

InTempo consists of two core operations, *Operationalization* and *Evaluation*, and the optional operation *Maintenance*. We outline each operation below.

Operationalization decomposes each of the input queries into a directed acyclic graph, i.e., a network, where nodes are simpler sub-queries and edges capture the ordering in which sub-queries should be evaluated. The operation is only performed once per query (at the beginning) and if the set of input queries has changed between invocations of InTempo.

Evaluation executes the network(s) over the RTM^H. The operation entails processing the changes made to the RTM^H since the last invocation of InTempo and evaluating the network nodes. The nodes store their results in-between executions and only update them if a change to the RTM^H affects them.

Maintenance prunes all deleted entities in the RTM^H which are irrelevant to future query evaluations. Query evaluations over a pruned RTM^H may yield faster answers while reducing the memory consumption; however, as the

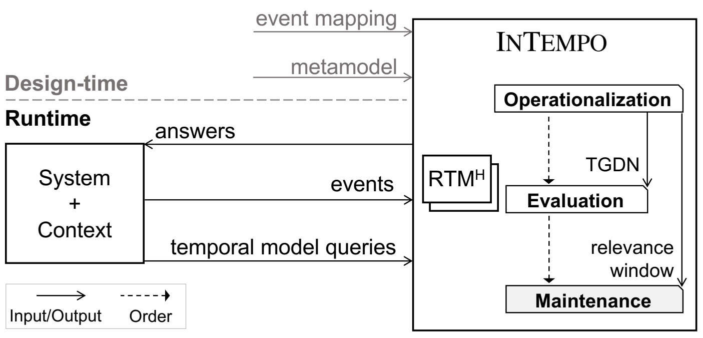


Figure 4.1: Overview of InTempo

duration of history encoded by the RTM^H is reduced, the temporal validity of matches refers to a restricted period of time.

Operationalization relies on a novel operationalization framework which builds on the framework from Section 2.2.4 to support queries in $\mathcal{L}_T$. The novel framework supports the construction of two types of networks: one where the terminal node of the network computes the satisfaction span $\mathcal{Z}$ of found matches for the MTGC of the query, and one where the terminal node computes the definite satisfaction span $\mathcal{Z}^d$ and the definite falsification span F. As definite computations involve the time point of the RTM^H, every new instance of an RTM^H calls for a re-computation of all definite spans; hence, in a network that computes $\mathcal{Z}^d$ and F, every new RTM^H instance calls for a re-execution of all network nodes such that all spans of stored matches are updated. This requirement, which does not affect a network that computes $\mathcal{Z}$, mitigates the degree of incrementality of the query evaluation. Aiming for highly incremental query evaluations, Operationalization uses only the method of the framework that constructs the network that computes $\mathcal{Z}$ and Evaluation is equipped with a technique that returns only those time points of $\mathcal{Z}$ that are definite; the caveat is that, for certain MTGCs, this technique returns answers with a certain delay.

Maintenance relies on the computation of a relevance window (during Operationalization) which is based on timing constraints of the temporal operators in input queries. Based on the window, the operation decides when deleted entities are not going to be involved in future query evaluations and can be thus pruned from the RTM^H.

We organize this chapter as follows. The framework used by Operationalization for the decomposition of queries into networks is presented in Section 4.1. The network execution, performed by Evaluation, and the technique that enables highly incremental query evaluation are presented in Section 4.2. Pruning, performed by Maintenance, and query evaluations over a pruned RTM^H are discussed in detail in Section 4.3. Section 4.4 presents a typical application scenario for InTempo: the integration of the approach operations with the well-known MAPE-K loop [93]. The integration serves as a reference adaptation engine for history-aware self-adaptation.

4.1 OPERATIONALIZATION

In the following, we refer to the operationalization framework in [23], summarized in Section 2.2.4, as *base approach*. The base approach considers graph queries in $\mathcal{L}$, i.e., where ACs are formulated as NGCs, and thus does not support temporal requirements on the evolution of patterns. Building on the base approach, we present a *temporal* approach which supports the construction of a *Temporal Generalized Discrimination Network* (TGDN) based on an AC formulated as an MTGC, thereby enabling the operationalization of queries in $\mathcal{L}_T$. The construction depends on the structure of the MTGC, therefore the structure of a TGDN for the satisfaction computation is identical to that of a TGDN for the definite computations. In the interest of simplicity, the following subsections focus on a TGDN for the satisfaction computation; we elaborate on the definite computations at the end of the section.

4.1.1 *Temporal Generalized Discrimination Network*

As in the base approach, we rely on *Graph Transformation* (GT) and consider the TGDN as directed acyclic graph consisting of non-deleting GT rules, called marking rules. Specifically, given a query $\zeta = (n, \psi)$ in $\mathcal{L}_{\mathrm{T}}$ with n and ψ typed over a type graph TG, a TGDN is a *Graph Transformation System* (GTS) characterized by the so-called *context pattern* n and a set of marking rules $\mathcal{R}$ induced by ψ. Patterns and morphisms in $\mathcal{R}$ are typed over an extended type graph TG', which contains all marking nodes and marking edges required by rules in $\mathcal{R}$.

In order to compute the satisfaction span $\mathcal{Z}$ of a sub-condition in ψ, the RHS of a rule $r \in R$ performs (interval) set operations based on the lifespans of matches for the LHS of r. The result of these operations constitutes the *duration* of the match being marked, and is stored in a distinguished attribute d of type interval set in the created marking node; the type of marking nodes in TG' are correspondingly equipped with this attribute.

As an MTGC may depend on a nested condition to compute its $\mathcal{Z}$, so may r depend on the duration of marking nodes created by its dependencies to compute the duration of its own marking node. This implies that the LHS of r has to be extended by marking nodes of dependencies, so that the duration of these marking nodes can be included in the computation performed by the RHS of r. These extensions to rules in $\mathcal{R}$ are novel compared to the base approach, where dependencies are included in the AC of a rule. The extensions characterize a dependency relation which should fulfill certain conditions. We define TGDNs and these conditions below.

Definition 4.1.1 (temporal generalized discrimination network)**.** Given a context pattern n typed over TG, an MTGC ψ over n also typed over TG, a set of marking rules $\mathcal{R}$ induced by ψ and typed over an extended type graph TG' which adds the required types for marking nodes and edges to TG, and a dependency relation consisting of rule pairs (r, r') from rules in $\mathcal{R}$ such that the marking node created in the RHS of r' is contained in the LHS of r, then the GTS $g = (\mathcal{R}, TG')$ is a *Temporal Generalized Discrimination Network* (TGDN) for ψ over n if the following conditions: (i) the transitive closure of the dependency relation is acyclic (ii) there is a unique rule called *root* on which no rule depends (iii) no two rules exist that directly depend on the same rule.

Applied at an RTM$^{\mathrm{H}}$ $H_{[\tau]}$, the root of the TGDN creates a marking node for each match m for n in $H_{[\tau]}$; the interval set assigned to the duration of each marking node is equal to the satisfaction span $\mathcal{Z}(m, \psi)$ of m for ψ.

We proceed by introducing Amalgamated Marking Rules in Section 4.1.2, which are required for the computation of the duration for certain operators of MTGL. We describe the rules induced by the operators in detail in Section 4.1.3. Then, in Section 4.1.4, we define a recursive operation that, given a query in $\mathcal{L}_{\mathrm{T}}$ constructs a TGDN according to Definition 4.1.1. The equivalence between the result of a TGDN and the set of all pairs $(m, \mathcal{Z}(m, \psi))$ is shown in Section 4.1.5, which also presents the method of obtaining the answer set $\mathcal{T}$ of a query based on the constructed TGDN. Section 4.1.6 elaborates on a TGDN for definite computations.

4.1.2 *Amalgamated Marking Rules*

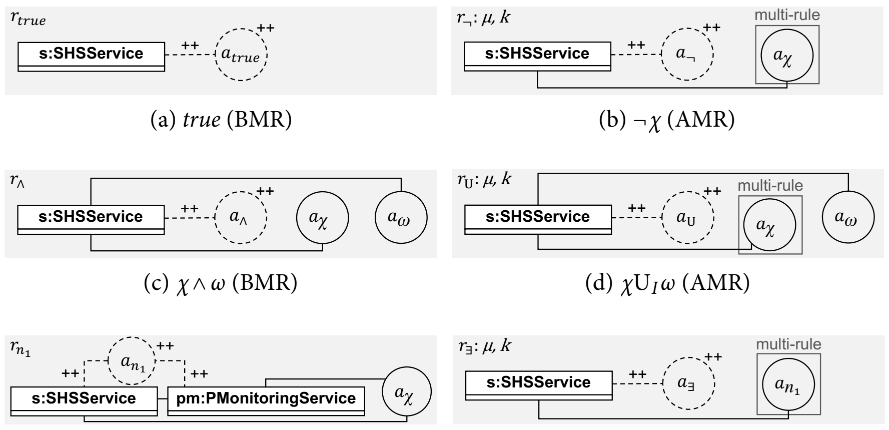
The computation of the duration of a marking node requires a certain functionality which conventional marking rules do not offer. For example, let $(n, \exists\hat{n})$ be a query in $\mathcal{L}_{\mathrm{T}}$. According to Definition 3.2.1, the computation of the satisfaction span $\mathcal{Z}$ for $\exists\hat{n}$ relies on the set of all matches for $\hat{n}$ that are compatible with a match m for n, i.e., the set $\hat{M}$ in Equation 3.5. Therefore, the marking rule induced by $\exists\hat{n}$ is required to keep track of all these matches and compute the union of their temporal validity. However, the number of these matches may vary in each application of the rule; the LHS of a conventional marking rule in the base approach cannot be adjusted before each application of the rule, depending on the number of found matches.

The computation of $\mathcal{Z}$ for the operators negation, *until*, and *since* contain similar conditionals. The satisfaction of *until* and *since* requires that the right operand is satisfied, however, when $0 \in I$, the satisfaction of the left operand is optional. The computation for negation is conditional on whether the negated MTGC is satisfied. For instance, assume the MTGC $\neg\exists\hat{n}$. The intuition of negation in an interval-based logic as MTGL is the following: If a match $\hat{m}$ for $\hat{n}$ exists, then negation is satisfied for the entire time domain minus the lifespan of $\hat{m}$; else, negation is satisfied for the entire time domain. The marking rule corresponding to negation should be applied both when $\hat{m}$ exists, whereby the LHS of the rule should include the marking node created by $\exists\hat{n}$, and when $\hat{m}$ doesn't exist, whereby the LHS should exclude that marking node. The definition and behavior of rules corresponding to these operators vary according to whether a marking node in the LHS has been matched.

Besides marking rules based on conventional GT rules used in the base approach, henceforth called *Basic Marking Rules* (BMRs), in order to support the varying behavior of rules described above, we employ amalgamated GT rules. In summary, amalgamated GT rules comprise a kernel rule and multi-rules: The kernel rule contains elements common to all rules and is to be applied only once; a multi-rule extends the kernel rule and may be applied arbitrarily many times. Matches of the multi-rule are *glued* at the match of their common kernel rule which induces the amalgamated GT rule—see Section A.2 for a more elaborate introduction. Based on amalgamated GT rules, we introduce the *Amalgamated Marking Rule* (AMR). Technically, when constructed, an amalgamated GT rule is a basic GT rule [25], thus, a TGDN which contains AMRs is compatible to the base approach. As explained previously, rules in the temporal approach, thus also AMRs, equip created marking nodes with an attribute d of type interval set, and the RHS of the rules is capable of performing a computation for assigning a value to d. In the context of a TGDN, an AMR is employed to group a varying number of marking nodes by a kernel match and perform a computation over the duration of these matched marking nodes, as required by MTGL operators.

4.1.3 *The Marking Rules Induced by Operators of MTGL*

The marking rules induced by MTGL operators in a TGDN are based on a context pattern. The context pattern corresponds to either the query pattern n



(a) *true* (BMR)

(b) $\neg\chi$ (AMR)

(c) $\chi \wedge \omega$ (BMR)

(d) $\chi \mathrm{U}_I \omega$ (AMR)

(e) $\exists(n_1, \chi)$ (a BMR and an AMR)

Figure 4.2: The rules induced by MTGL operators in the temporal approach. The type of rules is in parentheses. The assumed context pattern for all rules contains only the entity `s` of type `SHSService`; *exists* is based on n_1 in Figure 2.4. The rule induced by *since* is identical in structure to that for *until*.

or the pattern of an enclosing *exists* in an MTGC, e.g., given the query (n, ψ), the context pattern of ψ is initially n; if $\psi := \exists(\hat{n}, \chi)$, the context pattern of χ switches to $\hat{n}$. Rule definitions are illustrated in Figure 4.2, where the context pattern for all rules is an exemplary pattern which contains only the entity `s` of type `SHSService` and *exists* is based on pattern n_1 in Figure 2.4. For presentation purposes, the multi-rule and the kernel rule of an AMR are always depicted as a single rule.

All rules are defined over the extended type graph TG'. Moreover, a marking node a created by the RHS of a rule is accompanied by the creation of marking edges from a to all marked vertices as described in Section 2.2.4; for brevity, this step is omitted from the rule definitions below. Lastly, if a rule r is dependent on a rule r', we refer to the LHS of r that includes the marking node a created by the RHS of r' (see Definition 4.1.1) as *extended*. We denote an extended pattern by $\utilde{n}_{r'}$ or $\utilde{n}_\chi$, where χ is the MTGC that induced r'. If a rule has no dependency, then the LHS of that rule is not extended.

We elaborate on the rule induced by each MTGL operator below.

(i) *exists*: An *exists*, e.g., $\exists(\hat{n}, \chi)$, induces a BMR $r_{\hat{n}}$ and an AMR $r_\exists$. The BMR $r_{\hat{n}}$ is defined as $\langle p_{\hat{n}} : \utilde{\hat{n}}_\chi \hookrightarrow R_{\hat{n}}, \neg\exists p_{\hat{n}} \rangle$, where $\hat{n}_\chi = \hat{n}^V \uplus a_\chi$, i.e., the pattern $\hat{n}$ extended by the marking node a_χ created by the dependency of $r_{\hat{n}}$, which was induced by χ. The RHS is defined as $R_{\hat{n}} = \hat{n}^V \uplus a_{\hat{n}}$, i.e., it creates a marking node $a_{\hat{n}}$ that only marks the vertices of $\hat{n}$, omitting the marking node a_χ in $\utilde{\hat{n}}$. The AC of the rule requires that a match is not marked more than once.

The value of the duration $a_{\hat{n}}.d$ is assigned based on the lifespan λ of the match. Since matched elements may be marking nodes, the computation of λ is slightly adjusted compared to Equation 3.1; for a matched element e, its lifespan is given by the duration $e.d$ if e is a marking node, and by $[e.cts, e.dts)$ otherwise. As mentioned previously, the duration of a marking node created by a rule, corresponds to the satisfaction span of a match for the MTGC that induced the rule. The λ of a match for the LHS of $r_{\hat{n}}$, i.e., the pattern $\hat{\underset{\sim}{n}}_{\chi}$, is the intersection of (i) the $\lambda^{\hat{m}}$ of a match $\hat{m}$ for $\hat{n}$ and (ii) the duration $a_{\chi}.d$ of the marking node created by r_{χ}, i.e., the satisfaction span $\mathcal{Z}(\hat{m}, \chi)$; therefore, this intersection *corresponds to the temporal validity* $\mathcal{V}(\hat{m}, \chi)$.

Given the context pattern n and match m for n, the satisfaction span $\mathcal{Z}(m, \exists(\hat{n}, \chi))$ relies on the set of *all* matches for $\hat{n}$ that are compatible with a match m for n, i.e., the set $\hat{M}$ in Equation 3.5. In order to keep track of these matches, we accompany the rule $r_{\hat{n}}$ with an AMR. This AMR comprises a kernel rule k and a multi-rule μ.

The kernel rule k is defined as $k \coloneqq \langle p_k : n \hookrightarrow R_k, \neg\exists p_k \rangle$ with n the context pattern and $R_k = n^V \uplus a_{\exists}$ with $a_{\exists}$ the created marking node. The duration of $a_{\exists}$ is set to be the union of the duration of all instances of $a_{\hat{n}}$, i.e., all copies of the multi-rule, that are found during the construction of the AMR. This union computes a result equal to Equation 3.5.

The multi-rule μ is dependent on the BMR $r_{\hat{n}}$ and defined as $\mu \coloneqq \langle p_\mu : \underset{\sim}{n}_{r_{\hat{n}}} \hookrightarrow R_\mu, true \rangle$ with $\underset{\sim}{n}_{r_{\hat{n}}} = n^V \uplus a_{\hat{n}}$, i.e., the context pattern n extended by the marking node created by $r_{\hat{n}}$, and $R_\mu = \underset{\sim}{n}_{r_{\hat{n}}}$. The graph R_μ is identical to $\underset{\sim}{n}_{r_{\hat{n}}}$ because the rule is only required to track the matches for $\hat{n}$. The AC of μ is *true* as there is no marking node created and, therefore, no danger of marking a match twice. Moreover, matches for the LHS which include instances of $a_{\hat{n}}$ have to overlap in the match for the kernel rule, which ensures that only those matches will lead to a creation of a copy for the multi-rule during the construction of the AMR.

(ii) *negation*: The negation of an MTGC χ, i.e., $\neg\chi$, induces an AMR comprising a kernel rule k and a multi-rule μ. Given the context pattern n, the kernel rule is defined as $k \coloneqq \langle p_k : n \hookrightarrow R_k, \neg\exists p_k \rangle$ with $R_k = n^V \uplus a_{\neg}$. The multi-rule is defined as $\mu \coloneqq \langle p_\mu : \underset{\sim}{n}_{\chi} \hookrightarrow R_\mu, true \rangle$ with $\underset{\sim}{n}_{\chi} = n^V \uplus a_{\chi}$ and $R_\mu = \underset{\sim}{n}_{\chi}$. As all rule definitions that create a marking node require that a match is marked only once, for a given match of the kernel rule, the multi-rule can find either a single match or no match for the marking node a_{χ} during the construction of the AMR. The duration of $a_{\neg}$ is set according to Equation 3.3, i.e., $\mathbb{R} \setminus a_{\chi}.d$. If the marking node a_{χ} has not been created, i.e., if χ is never satisfied, and a match for the kernel rule exists, the kernel rule is still applied. In that case, the duration of $a_{\neg}$ is computed by $\mathbb{R} \setminus \varnothing = \mathbb{R}$.

(iii) *conjunction*: The marking rule induced by the MTGC $\chi \wedge \omega$ is a BMR as, in this case, both operands must be satisfied, i.e., both marking nodes of rules corresponding to the operands must be matched, in order for the conjunction to be satisfied—see Equation 3.4. The rule $r_{\wedge}$ is defined as $r_{\wedge} \coloneqq \langle p_{\wedge} : \underset{\sim}{n}_{\chi,\omega} \hookrightarrow R_{\wedge}, \neg\exists p_{\wedge} \rangle$ with $\underset{\sim}{n}_{\chi,\omega} = n^V \uplus \{a_{\chi}, a_{\omega}\}$ and $R_{\wedge} = n^V \uplus a_{\wedge}$. The duration of $a_{\wedge}$ is set according to Equation 3.4.

(iv) *until*: The MTGC $\chi \mathrm{U}_I \omega$ induces an AMR with a kernel rule k and one multi-rule μ_{χ}. The kernel rule includes a marking node corresponding to the right operand, i.e., a rule r_{ω}, thus requiring that the right operand is satisfied, as this is necessary for *until* to be satisfied. The existence of a marking node for the left operand, i.e., a rule r_χ may be optional, i.e., when $0 \in I$, whereby *until* is satisfied regardless of the satisfaction of the left operand. The kernel rule is defined as $k \coloneqq \langle p_k : \underset{\sim}{n}_\omega \hookrightarrow R_k, \neg\exists p_k \rangle$ with $\underset{\sim}{n}_\omega = n^V \uplus a_\omega$ and $R_k = n^V \uplus a_{\mathrm{U}}$. The duration of a_{U} is set according to Equation 3.6. The multi-rule is defined as $\mu_\chi \coloneqq \langle p_{\mu_\chi} : \underset{\sim}{n}_\chi \hookrightarrow R_{\mu_\chi}, true \rangle$ with $\underset{\sim}{n}_\chi = n^V \uplus a_\chi$ and $R_{\mu_\chi} = \underset{\sim}{n}_\chi$. The multi-rule μ_χ can find either a single match or no match for the marking node v_χ during the construction of the AMR. Regardless of whether a marking node for χ is found, if a match for $\underset{\sim}{n}_\omega$ exists, the kernel rule is still applied.

(v) *since*: The marking rule induced is identical to the marking rule for *until*, except the duration of the marking node is set according to Equation 3.7.

(vi) *true*: According to the semantics of NGCs and, in turn, MTGL, the operator *true* is always satisfied. Therefore, in the base approach, the operator does not induce a marking rule; instead, it is implicitly mapped to an AC or, rather, the lack of an additional AC, in a marking rule which searches for a pattern. In the temporal approach however, the operator may be involved in the computations of the satisfaction span of other operators, i.e., *once* or *eventually*, which explicitly require the presence of a marking node with a duration. Thus, in the temporal approach the operator *true* induces a BMR, which is defined as $r_{true} \coloneqq \langle p_{true} : n \hookrightarrow R_{true}, \neg\exists p_{true} \rangle$ with n the context pattern and $R_{r_{true}} = n^V \uplus a_{true}$. The duration of the marking node is equal to $\mathbb{R}$, as in Equation 3.2.

The involvement of a_{true} in the LHS of a BMR induced by an *exists*, e.g., $\exists(n, true)$—see item (i), has virtually no effect: the BMR r_n induced by $\exists(n, true)$, computes the duration of the marking node it creates by intersecting the lifespans of all elements in the match for the LHS; therefore, the computation for the pattern $\underset{\sim}{n}_{true}$ will be equivalent to $\lambda^m \cap \mathbb{R} = \lambda^m$, with m a match for n. However, the involvement of a_{true} in other operators, e.g., *once* or *eventually*, does have an effect on the result.

4.1.4 *TGDN Construction*

We now define the operation $\mathcal{C}$, which, given a query in $\mathcal{L}_{\mathrm{T}}$, obtains the set of rules $\mathcal{R}$ and the marking type graph TG'. The operation uses these characteristics to construct a TGDN. The output of $\mathcal{C}$ also includes the root r of the constructed TGDN. The following definition refers to items from Section 4.1.3.

Definition 4.1.2 (construct a TGDN (Operation $\mathcal{C}$))**.** The operation $\mathcal{C}$ takes a query (n, ψ) as input; ψ is an MTGC over n and both n and ψ are typed over a type graph TG. The operation returns a tuple (g, r) with g a TGDN $g = (\mathcal{R}, TG')$ and r the root of g.

As a preliminary step, the operation traverses ψ and constructs the marking type graph TG'. The operation proceeds by performing the following recursive operation $\mathcal{C}_{rec}(n, \psi)$ which populates $\mathcal{R}$ and returns r:

- $\psi = true$ and
  - A BMR r_{true} is created according to item (vi) and added to $\mathcal{R}$. The LHS of the pattern is the input pattern n.

  - The rule r_{true} is returned.

- $\psi = \neg\chi$ and
  - The dependency $r_\chi = \mathcal{C}_{rec}(n, \chi)$ is obtained.
  - An AMR $r_\neg$, i.e., a kernel rule k and a multi-rule μ, is created according to item (ii). The pattern of μ includes the marking node created by r_χ. Both rules are added to $\mathcal{R}$.
  - The rule k is returned.

- $\psi = \chi \wedge \omega$ and
  - The dependencies $r_\chi = \mathcal{C}_{rec}(n, \chi)$ and $r_\omega = \mathcal{C}_{rec}(n, \omega)$ are obtained.
  - A BMR $r_\wedge$ is created according to item (iii) and added to $\mathcal{R}$. The pattern of $r_\wedge$ includes the marking node of r_χ and r_ω.
  - The rule $r_\wedge$ is returned.

- $\psi = \exists(\hat{n}, \chi)$ and
  - The dependency $r_\chi = \mathcal{C}_{rec}(\hat{n}, \chi)$ is obtained.
  - A BMR $r_{\hat{n}}$ is created according to item (i), whose LHS includes the marking node created by r_χ. An AMR $r_\exists$, comprising a kernel k and a multi-rule μ, is also created according to the same item. All created rules are added to $\mathcal{R}$.
  - The rule k is returned.

- $\psi = \chi \mathrm{U}_I \omega$ and
  - The dependencies $r_\chi = \mathcal{C}_{rec}(n, \chi)$ and $r_\omega = \mathcal{C}_{rec}(n, \omega)$ are obtained.
  - An AMR r_U, i.e., a kernel rule k and a multi-rule μ, is created according to item (iv). The pattern of μ_χ includes the marking node created by r_χ, whereas the pattern of k includes the marking node created by r_ω. Both rules are added to $\mathcal{R}$.
  - The rule k is returned.

- $\psi = \chi \mathrm{S}_I \omega$ and the operation proceeds similarly to *until*, although the rules are created according to item (v).

Finally, based on the obtained TG', the obtained $\mathcal{R}$, and the r returned by $\mathcal{C}_{rec}(n, \psi)$, the operation constructs the TGDN g and returns the tuple (g, r).

The structure of an MTGC and the construction of a TGDN by the operation $\mathcal{C}$ ensures that the network satisfies the conditions in Definition 4.1.1.

Example 4.1.1 (TGDN construction)**.** *See Figure 4.3 for the TGDN g_1 constructed by $\mathcal{C}(n_1, \psi_1)$ with $\psi_1 := \neg\exists\, n_{1.1}\, \mathrm{U}_{[0,60]}\, \exists\, n_{1.2}$. The patterns $n_1, n_{1.1}$, and $n_{1.2}$ are depicted in Figure 2.4. As the MTGC $\exists\, n_{1.1}$ is an abbreviation for $\exists(n_{1.1}, \mathrm{true})$, it induces three rules: a BMR r_{true_1} for the MTGC* true*; a BMR $r_{n_{1.1}}$ and an AMR $r_{\exists_1}$ for* exists*. The rule $r_{n_{1.1}}$ depends on r_{true_1}. The LHS of r_{true_1} is the pattern $n_{1.1}$ and the RHS of the rule creates a marking node of type a_{true_1}. The LHS of $r_{n_{1.1}}$ is the pattern $n_{1.1}$ extended by the marking node a_{true_1} created by r_{true_1}; the RHS of the rule creates a marking node of type $a_{n_{1.1}}$.*

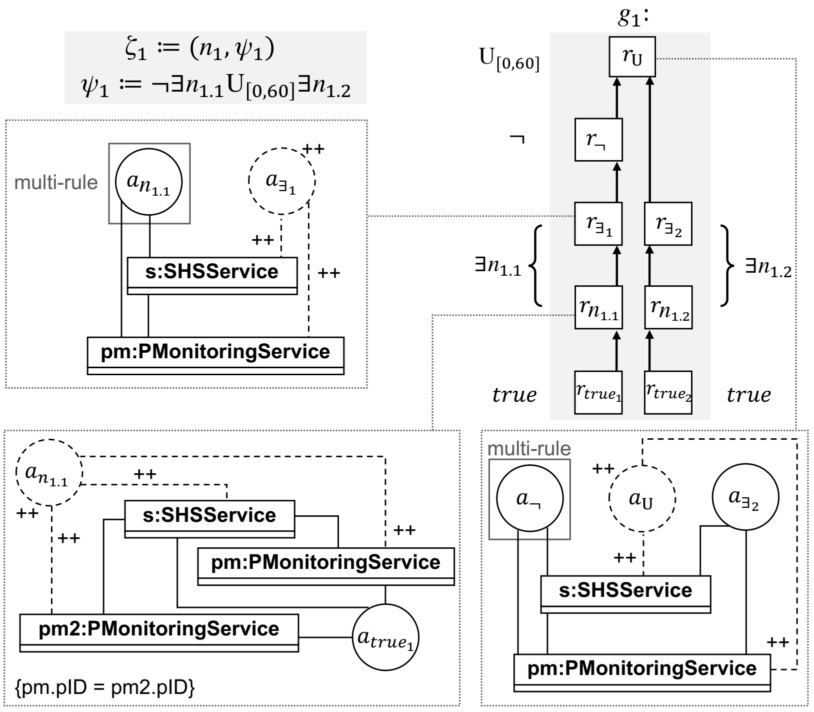


Figure 4.3: The TGDN constructed by $\mathcal{C}(n_1, \psi_1)$ with $\psi_1 \coloneqq \neg\exists\, n_{1.1}\, \mathrm{U}_{[0,60]}\, \exists\, n_{1.2}$, where AMRs, i.e., r_{U}, $r_{\neg}$, $r_{\exists_1}$, and $r_{\exists_2}$, are represented by a single rule; the context pattern in the AMRs is the pattern n_1 from Figure 2.4.

The rule $r_{\exists_1}$ comprises a multi-rule and a kernel rule. The multi-rule depends on $r_{n_{1.1}}$. The LHS of the multi-rule tracks marking nodes of type $a_{n_{1.1}}$ created by $r_{n_{1.1}}$. The RHS of the rule creates no marking node. The LHS of the kernel rule is the context pattern n_1, i.e., a vertex of type `SHSService` and a vertex of type `PMonitoringService`.The RHS of the kernel rule creates a marking node of type $a_{\exists_1}$. Similarly to $\exists\, n_{1.1}$, three rules r_{true_2}, $r_{n_{1.2}}$, $r_{\exists_2}$ are created for $\exists\, n_{1.2}$.

The MTGC $\exists\, n_{1.1}$ is enclosed by a negation, which induces an AMR $r_{\neg}$. The multi-rule of $r_{\neg}$ depends on $r_{\exists_1}$. The multi-rule tracks marking nodes of type $a_{\exists_1}$. The LHS of the kernel rule is the context pattern n_1. The RHS of the kernel rule creates a marking node of type $a_{\neg}$.

Finally, the root of g_1 is the AMR induced by the until *in ψ_1. The multi-rule tracks marking nodes of type $a_{\neg}$ created by $r_{\neg}$. The LHS of the kernel rule is the context pattern n_1 extended with the marking node of the RHS of the* until *operator, i.e., the marking node $a_{\exists_2}$ created by $r_{\exists_2}$.*

4.1.5 *Obtaining the Temporal Validity*

The root of a TGDN g for ψ over n executed over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$ finds all matches for n and computes their respective satisfaction span for ψ. Conceptually, we refer to this result as the *satisfaction span set* and define it as follows.

Definition 4.1.3 (satisfaction span set Σ)**.** Given an MTGC ψ over n, the *satisfaction span set* Σ_ψ of ψ over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$ is given by:

$$\Sigma_\psi(H_{[\tau]}) = \{(m, \mathcal{Z}(m,\psi)) | m \text{ is a match for } n\}$$

A satisfaction span $\mathcal{Z}(m,\psi)$ in Σ_ψ can be empty.

We now define the *result of a TGDN* in technical terms. The TGDN result $\mathcal{G}$ is the result of the execution of a TGDN over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$, i.e., the application of the rules in $\mathcal{R}$ to $H_{[\tau]}$. In the result $\mathcal{G}$, we make use of an operation $obt(a)$, which, given a marking node a, it obtains the match marked by a. For instance, assume that a marking rule r searches for a pattern n. Let m be a match for the LHS of r and a be the marking node marking the match m after the application of RHS of r. The operation $obt(a)$ obtains the match m without the marking node and marking edges.

Definition 4.1.4 (TGDN result $\mathcal{G}$)**.** Given a TGDN g for an MTGC ψ over a context pattern n, with r the root of g, the *TGDN result* $\mathcal{G}_\psi$ of g over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$ is given by:

$$\mathcal{G}_\psi(H_{[\tau]}) = \{(obt(a), a.d) | a \text{ is a marking node created by } r\}$$

As before, $a.d$ is the duration of a. To correspond to the satisfaction span set, the duration $a.d$ in $\mathcal{G}$ can be empty.

We now show that given a match m for the context pattern n, the execution of a TGDN for ψ over n, i.e., the application of the rules in g, creates a marking node a for m and the duration $a.d$ of this marking node is equal to the satisfaction span of m for ψ, i.e., $\mathcal{Z}(m,\psi)$.

Lemma 4.1.1 (Equality of satisfaction span set and TGDN result)**.** *Given an RTM^H $H_{[\tau]}$, a context pattern n, an MTGC ψ over n, and a TGDN g characterized by the output of $\mathcal{C}(n,\psi)$, the satisfaction span set Σ_ψ in $H_{[\tau]}$ is equal to the result $\mathcal{G}_\psi$ of g, that is:*

$$\Sigma_\psi(H_{[\tau]}) = \mathcal{G}_\psi(H_{[\tau]})$$

Proof (idea). By structural induction over ψ and the TGDNs that operation $\mathcal{C}(n,\psi)$ constructs. See Section A.3.5 for the proof. □

In the temporal approach, the construction of a TGDN g for ψ of a query (n,ψ) is always followed by the extension of the network by a distinguished rule, called *terminal*, induced by the query pattern n. The terminal rule depends on the root of g and enables the TGDN to compute the temporal validity $\mathcal{V}$ of a match; thus, this extended network comprising the TGDN g and the terminal rule is called the *Temporal Validity Generalized Discrimination Network* (TVGDN). The TVGDN returns a result that is equivalent to the query answer set $\mathcal{T}$. We define the terminal rule together with its result below.

Definition 4.1.5 (terminal rule r_t, TVGDN $g^{\mathcal{V}}$, TVGDN result $\mathcal{G}^{\mathcal{V}}$)**.** Given a query (n,ψ) in $\mathcal{L}_{\mathrm{T}}$, a TGDN g constructed by $\mathcal{C}(n,\psi)$, and r the root of g, the *terminal rule* r_t is a BMR whose LHS is the pattern $\underset{\sim}{n}_r$, i.e., the pattern n extended with the marking node created by r.

The *Temporal Validity Generalized Discrimination Network* (TVGDN) $g^{\mathcal{V}}$ is obtained by an appropriate update to the marking type graph TG' and an addition of r_t to $\mathcal{R}$. The *TVGDN result* $\mathcal{G}^{\mathcal{V}}$ of $g^{\mathcal{V}}$ is given by:

$$\mathcal{G}_\psi^{\mathcal{V}}(H_{[\tau]}) = \{(obt(a), a.d) | a \text{ is a marking node created by } r_t \text{ and } a.d \neq \varnothing\}$$

Contrary to $\mathcal{G}$, $\mathcal{G}^{\mathcal{V}}$ admits no marking nodes with an empty duration. Following the addition of a terminal rule to a TGDN, the network continues to satisfy the conditions in Definition 4.1.1.

Example 4.1.2 (terminal rule, TVGDN). *Given the query $\zeta_1 := (n_1, \psi_1)$ and the TGDN g_1 constructed by $\mathcal{C}(n_1, \psi_1)$ in Example 4.1.1, the terminal rule r_{t1} for g_1 is a BMR whose LHS is the pattern n_1 extended by the marking node v_{U} created by r_{U}, i.e., the root of g_1.*

The TVGDN $g_1^{\mathcal{V}}$ can be obtained by appropriately adjusting the marking type graph TG' and adding r_{t1} to $\mathcal{R}$ of g_1.

Finally, we show that the result $\mathcal{G}^{\mathcal{V}}$ of a TVGDN is equal to the query answer set $\mathcal{T}$ (see Definition 3.2.3) of a query in $\mathcal{L}_{\text{T}}$.

Theorem 4.1.1 (equality of TVGDN result and query answer set). *Given an RTM^H $H_{[\tau]}$, a query (n, ψ) in $\mathcal{L}_{\text{T}}$, and a TVGDN $g^{\mathcal{V}}$ for (n, ψ), the result $\mathcal{G}^{\mathcal{V}}$ of $g^{\mathcal{V}}$ is equal to the query answer set $\mathcal{T}$ of (n, ψ) over $H_{[\tau]}$, that is:*

$$\mathcal{T}(H_{[\tau]}) = \mathcal{G}_{\psi}^{\mathcal{V}}(H_{[\tau]})$$

Proof. By showing inclusion in both directions.

Let t be a tuple $(m, \mathcal{V})$ in $\mathcal{T}$. By definition, $\mathcal{V}$ is not empty. The existence of t implies that there is also a tuple $t' = (m, \mathcal{Z}) \in \Sigma_{\psi}$, with $\mathcal{Z}$ being not empty. Recall that $g^{\mathcal{V}}$ is obtained by extending g with r_t. As shown by Lemma 4.1.1, the result $\mathcal{G}_{\psi}$ of g is equal to Σ_{ψ}. Therefore, there is a tuple $t'' = (obt(a), a.d)$ in $\mathcal{G}_{\psi}$ such that $t'' = t'$, i.e., a is a marking node created by the root r of g for a match m and $a.d$ is equal to the non-empty $\mathcal{Z}(m, \psi)$. The terminal rule r_t is dependent on r, therefore its LHS is $\underset{\sim}{n}_r$. As the match m for n exists and so does the marking node a created by r, r_t finds a match $\underset{\sim}{m}_r$ and computes the duration of the marking node a_t by intersecting the lifespans of all elements of $\underset{\sim}{m}_r$, i.e., including the marking node a created by r. This is equivalent to $\lambda^m \cap \mathcal{Z}(m, \psi)$, that is, the temporal validity of m. The terminal rule result $\mathcal{G}^{\mathcal{V}}$ contains $obt(a_t)$, i.e., the match m, paired with the non-empty duration $a_t.d$. Thus, it contains a tuple t''' such that $t''' = t$.

We proceed with the inverse direction. Let u be a tuple $(obt(a), a.d) \in \mathcal{G}^{\mathcal{V}}$. As the LHS of r_t is $\underset{\sim}{n}_r$, this implies that there is a match m for n. As shown above, the duration $a.d$ is equal to $\mathcal{V}_m$. Note that $\mathcal{G}^{\mathcal{V}}$ admits only marking nodes whose duration is not empty, therefore $a.d \neq \emptyset$. By definition of $\mathcal{T}$, there exists also a tuple $u' \in \mathcal{T}$ involving m, such that $u' = u$. □

4.1.6 *TGDN for Definite Computations*

The definite satisfaction span $\mathcal{Z}^d$ and definite falsification span F are interdependent. This implies that the nodes of a TGDN for definite computations have to perform both computations. Thus, assuming that the attribute d captures $\mathcal{Z}^d$, another duration attribute d^F which captures F has to be added to the typing of rules.

Since the construction of a TGDN depends on the MTGC structure, the structure of a TGDN for $\mathcal{Z}^d$ and F is identical to that of a TGDN for $\mathcal{Z}$; rule definitions in Section 4.1.3 only have to be adjusted so that the values of d and

d^F are set according to the definite computations in Section 3.3.3. Assuming that a TGDN g^d has been constructed by a construction operation $\mathcal{C}^d$ that uses these adjusted rule definitions and is otherwise the same as $\mathcal{C}$, we show equivalence between a definite answer set $\mathcal{T}^d$ (see Definition 3.3.4) and the result of a g^d analogously to Section 4.1.5.

Definition 4.1.6 (definite satisfaction and falsification span set Σ^d)**.** Given an MTGC ψ over n, the *definite satisfaction and falsification span set* Σ^d_ψ of ψ over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$ is given by:

$$\Sigma^d_\psi(H_{[\tau]}) = \{(m, \mathcal{Z}^d(m,\psi), F(m,\psi)) | m \text{ is a match for } n\}$$

The definite spans $\mathcal{Z}^d(m,\psi)$, $F(m,\psi)$ in Σ^d_ψ can be empty.

Definition 4.1.7 (TGDN definite result $\mathcal{G}^d$)**.** Given a TGDN g^d for an MTGC ψ over a context pattern n, with r the root of g^d, the *definite TGDN result* $\mathcal{G}^d_\psi$ of g^d over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$ is given by:

$$\mathcal{G}^d_\psi(H_{[\tau]}) = \{(obt(a), a.d, a.d^F) | a \text{ is a marking node created by } r\}$$

The values of $a.d$, $a.d^F$ capture the definite satisfaction span and definite falsification span, respectively. To correspond to Σ^d_ψ, $a.d$ and $a.d^F$ can be empty.

We now show that given a match m for the context pattern n, the execution of a TGDN g^d for ψ over n creates a marking node a for m and (i) $a.d$ is equal to $\mathcal{Z}^d(m,\psi)$ (ii) $a.d^F$ is equal to $F(m,\psi)$.

Lemma 4.1.2 (equality of definite satisfaction and falsification span set and TGDN definite result)**.** *Given an RTM^H $H_{[\tau]}$, a context pattern n, an MTGC ψ over n, and a TGDN g^d characterized by the output of $\mathcal{C}^d(n,\psi)$, the definite satisfaction and falsification span set Σ^d_ψ in $H_{[\tau]}$ is given by the definite result $\mathcal{G}^d_\psi$ of g^d, that is:*

$$\Sigma^d_\psi(H_{[\tau]}) = \mathcal{G}^d_\psi(H_{[\tau]})$$

Proof (idea). The proof proceeds similarly to Lemma 4.1.1, i.e., by structural induction over ψ and the TGDNs that operation $\mathcal{C}^d(n,\psi)$ constructs. See Section A.3.6 for the complete proof. □

As in Definition 4.1.4, g^d is extended by a terminal rule r^d_t, i.e., a BMR adjusted to compute a value for both d and d^F. Following an appropriate update to the extended type graph TG' and an addition of r^d_t to $\mathcal{R}$, this extension yields a TVGDN $g^{d,\mathcal{V}}$ whose result is equivalent to the definite answer set.

Definition 4.1.8 (TVGDN definite result $\mathcal{G}^{d,\mathcal{V}}$)**.** Given a query (n,ψ) in $\mathcal{L}_{\mathrm{T}}$ and a TVGDN $g^{d,\mathcal{V}}$ constructed by $\mathcal{C}^d(n,\psi)$ and extended by a terminal rule r^d_t, the *TVGDN result* $\mathcal{G}^{d,\mathcal{V}}$ of $g^{d,\mathcal{V}}$ over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$ is the set of all triples $(obt(a), a.d, a.d^F)$ such that (i) a is a marking node created by r_t and (ii) $(a.d \neq \emptyset) \vee (a.d^F \neq \emptyset)$.

Finally, we show that the result $\mathcal{G}^{\mathcal{V}}$ of a TVGDN is equal to the definite query answer set $\mathcal{T}^d$ (see Section 3.3.4) of a query in $\mathcal{L}_{\mathrm{T}}$.

Theorem 4.1.2 (equality of TVGDN definite result and query definite answer set)**.** *Given an RTM^H $H_{[\tau]}$, a query (n,ψ) in $\mathcal{L}_{\mathrm{T}}$, and a TVGDN $g^{d,\mathcal{V}}$ for (n,ψ), the result $\mathcal{G}^{d,\mathcal{V}}$ of $g^{d,\mathcal{V}}$ is equal to the definite query answer set $\mathfrak{T}^d$ of (n,ψ) over $H_{[\tau]}$, that is:*

$$\mathfrak{T}^d(H_{[\tau]}) = \mathcal{G}^{d,\mathcal{V}}_{\psi}(H_{[\tau]})$$

Proof (idea). The proof proceeds similarly to Theorem 4.1.1. See Section A.3.7 for the complete proof. □

4.2 QUERY EVALUATION

Evaluation in InTempo (see Figure 4.1) performs the evaluation of the query based on the TVGDN constructed by Operationalization. On a formal level, the evaluation entails applying the rules of the network *in batch* to a given RTM$^{\mathrm{H}}$ instance H. Given a new version of the RTM$^{\mathrm{H}}$ instance H', all rules would have to be reapplied; although marking nodes created in H would remain in H' and rule definitions prohibit the duplication of marking nodes, all previously marked matches would have to be re-matched before the AC of the rule could conclude that they have been already marked. This evaluation method is inefficient for repetitive query evaluations over a sequence of RTM$^{\mathrm{H}}$ instances.

Focusing on the implementation level, Evaluation applies practical optimizations which enable incremental query evaluation and are not covered by formal semantics of GT, i.e., the operation processes only the changes made to RTM$^{\mathrm{H}}$ since the last network execution; it only reapplies rules if a change concerns their LHS or the LHS of one of their dependencies; the nodes of a network store their results in-between executions and incrementally update them. Since Evaluation is the operation where the pattern matching is performed, another optimization which allows for faster query evaluations is that rules start the pattern matching effort in the surrounding of elements affected by the change—see local search in Section 2.2.5. Finally, as discussed in the beginning of this chapter, definite computations reduce the degree of incrementality in query evaluations; Evaluation is equipped with a technique to obtain an *effective answer set* which, although based on the (non-definite) satisfaction computation, contains definite answers. For certain MTGCs however, these answers are returned with a delay.

In the following, we demonstrate a batch query evaluation in Section 4.2.1—the batch evaluation occurs at the beginning of sequences. We demonstrate incremental query evaluation in Section 4.2.2. We discuss the effective answer set in detail in Section 4.2.3.

4.2.1 *Batch Query Evaluation*

In the following example, we demonstrate a batch query evaluation over an RTM$^{\mathrm{H}}$ instance which we assume is the first member of a sequence.

Example 4.2.1 (batch query evaluation)**.** *We demonstrate a batch evaluation of a query via an example based on the TVGDN $g_1^{\mathcal{V}}$ constructed for the query ζ_1 (see Example 4.1.1 and Example 4.1.2) and the RTM^H instance $H_{[5]}$ in Figure 3.2. For*

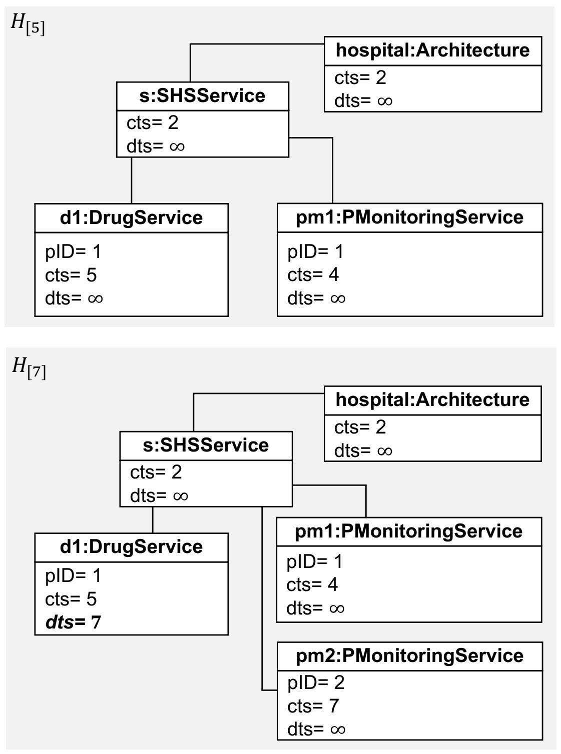


Figure 4.4: RTM^{H} Instances $H_{[5]}$ and $H_{[7]}$

convenience, the instance is also shown in Figure 4.4. Each item in the following lists marks the application of a marking rule in $g_1^{\mathcal{V}}$. Each marking rule application leads to the creation of marking nodes whose duration computation depends on the operator that induced the rule—see Section 4.1.3. As a reminder, exists *induces a BMR and an AMR, negation induces an AMR,* true *induces a BMR, and* until *induces an AMR. The network execution is recursive and starts with one of the leaves of $g_1^{\mathcal{V}}$.*

r_{true_1} *The LHS of the rule is the pattern $n_{1.1}$. No matches for $n_{1.1}$ are found, therefore no marking is created.*

$r_{n_{1.1}}$ *The LHS of the rule is the pattern $n_{1.1}$ extended by the marking node v_{true_1}, i.e., the marking node type created by r_{true_1}. No matches are found.*

$r_{\exists_1}$ *The multi-rule is applied first. The multi-rule finds no matches of marking nodes of the dependency of $r_{\exists_1}$, i.e., the rule $r_{n_{1.1}}$. Therefore, no copy of the multi-rule is created. The LHS of the kernel rule is the context pattern n_1. The vertices* `s` *and* `pm1` *are matched. The duration $a_{\exists_1}.d$ of the marking node $a_{\exists_1}$ created by the kernel rule is the union of all multi-rule copies. Since no copies exist, this union is empty. Hence, the empty interval is stored into $a_{\exists_1}.d$.*

$r_\neg$ *The multi-rule is applied first. The multi-rule finds one marking node created by the dependency, i.e., the node $a_{\exists_1}$. Therefore, a copy of the multi-rule is created. The LHS of the kernel rule is n_1. The kernel rule finds one match (vertices* `s` *and* `pm1`*). The copy of the multi-rule is glued into the kernel rule. The rule $r_\neg$ creates one marking node whose duration is computed based on the duration $a_{\exists_1}$: $\mathbb{R} \setminus a_{\exists_1}.d = \mathbb{R} \setminus \emptyset = \mathbb{R}$.*

The rule $r_\neg$ is one of the two dependencies of the rule r_U. Before proceeding, the rule r_U requires that the other dependency has been applied as well, hence, via recursion, the evaluation proceeds with the rule r_{true_2}:

r_{true_2} *The LHS of the rule is the pattern $n_{1.2}$. One match is found, containing* `s`, `pm1`, `d1`, *and the edges among them. One marking node a_{true_2} is created with duration $\mathbb{R}$.*

$r_{n_{1.2}}$ *The LHS of the rule is the pattern $n_{1.2}$ extended with the marking node a_{true_2}, i.e., the marking node type created by r_{true_2}. One match is found. A marking node $a_{n_{1.2}}$ is created whose duration is computed by intersecting all matched vertices. The computed duration is $[5, \infty)$, which coincides with the temporal validity of the match for the MTGC* true.

$r_{\exists_2}$ *The multi-rule is applied first. It finds one match for the marking node of its dependency, i.e., the marking node a_{true_2}, thus one copy of the multi-rule is created. The LHS of the kernel rule is the context pattern n_1. The vertices* `s` *and* `pm1` *are matched. The duration $a_{\exists_2}.d$ of the marking node $a_{\exists_2}$ created by the kernel rule is the union of all multi-rule copies. Based on the duration of $a_{n_{1.2}}$, this computation is equal to $[5, \infty)$.*

Since both of its dependencies have been applied, the rule r_U can now be applied:

r_U *The multi-rule is applied, which searches for marking nodes created by the rule induced by the left operand of* until, *i.e., the rule $r_\neg$. One marking node is found and a copy of the multi-rule is created. The LHS of the kernel rule is the pattern n_1 extended by the marking node created by the rule induced by the right operand, i.e., the rule $r_{\exists_2}$. One match is found containing the vertices* `s`, `pm1`, *and $a_{n_{1.2}}$. The copy of the multi-rule, i.e., the node $a_\neg$ is also glued into the kernel rule. The kernel rule proceeds with the computation of the duration: It first checks whether intervals stored in the duration of marking nodes of the right operand are adjacent or overlapping with any of the intervals in the duration of the of marking nodes of the left operand. In this case, this check is done for the duration of the nodes $a_{n_{1.2}}.d = [5, \infty)$ and $a_\neg.d = \mathbb{R}$. These intervals are indeed overlapping. The duration $a_\mathrm{U}.d$ of the marking node created by r_U is computed according* Equation 3.6 *to $[-55, \infty)$.*

r_{n_1} *The LHS of the rule is the pattern n_1 extended with the marking node type created by r_U. One match is found containing* `s`*,* `pm1`*, and a_U. The duration of the marking node a_{n_1} to be created is computed by intersecting the duration of each matched vertex. The vertices* `s` *and* `pm1` *effectively constitute a match m_1 for n_1, therefore intersecting their lifespans computes the lifespan λ^{m_1} of m_1; the duration $a_U.d$ = corresponds to $(m_1, \mathcal{Z}(m_1, \psi_1))$. Therefore, intersecting the lifespans of all matched vertices computes the temporal validity $\mathcal{V}$ of m_1. In this case, this validity is computed to be $[4, \infty) \cap [-55, \infty) = [4, \infty)$ and stored in the duration $a_{n_1}.d$ of the marking node a_{n_1} created by r_{n_1}.*

The evaluation returns a $\mathcal{G}^{\mathcal{V}}$ which contains $obt(v_{n_1})$, i.e., the match m_1 paired with the duration $a_{n_1}.d$, i.e., the temporal validity, $[4, \infty)$ of m_1, during which, besides being structurally present in the graph, the match satisfies the temporal requirements specified in ψ_1.

4.2.2 *Incremental Query Evaluation*

We demonstrate an incremental query evaluation, which always follows a previous query evaluation. The Evaluation operation of InTempo collects the changes that are made to the RTM$^{\mathrm{H}}$ since the previous evaluation and, for each change, triggers a new incremental evaluation.

The application of rules in an incremental query evaluation may entail the re-computation of the duration of marking nodes that have been created in a previous evaluation. Although on an implementation level such applications can be performed directly during the query evaluation, they would require intermediate steps on a formal level, which are not covered by the framework in Section 4.1.

In the following example, we demonstrate an incremental query evaluation, which we assume is performed following the query evaluation in Example 4.2.1.

Example 4.2.2 (incremental query evaluation)**.** *Example 4.2.1 demonstrated the batch evaluation of query ζ_1 over $H_{[5]}$. The changes occurring at time point 7 yield a new RTM^H instance, i.e., $H_{[7]}$—see Figure 4.4. The changes entail the deletion of* `d1`*, and the creation of* `pm2` *as well as of an edge between* `pm2` *and* `s`*. In practice, each change triggers a new query evaluation; however, in the interest of brevity, the example below processes all changes occurring at time point 7 in a single evaluation. Consequently, changes affect the types of entities in all the LHSs of all rules and, hence, all rules are reapplied. In this example, rule applications perform the computations discussed during the presentation of the satisfaction computation in Example 3.2.1. The evaluation proceeds as follows.*

r_{true_1} *No match is found.*

$r_{n_{1.1}}$ *No match is found.*

$r_{\exists_1}$ *The multi-rule finds no marking nodes created by r_{true_1}. A new match for the LHS of the kernel exists (the one involving `pm2`) whose duration is also the empty interval. The match found in the previous evaluation remains unchanged.*

$r_{\neg}$ *The multi-rule finds no marking nodes created by $r_{n_{1.1}}$. The LHS of the kernel rule is the pattern n_1. Thus, a new match is found for `s` and `pm2` whose duration is computed to be $\mathbb{R}$, similarly to the previous evaluation over $H_{[5]}$. The match found in the previous evaluation remains unchanged.*

r_{true_2} *No new matches are found.*

$r_{n_{1.2}}$ *No new matches are found, however, due to the deletion of `d1`, the duration of the marking node a_{true_2} found in the previous evaluation is updated to $[5,7)$.*

$r_{\exists_2}$ *Since the duration a_{true_2} has been changed, a copy of the multi-rule for a_{true_2} is created. This copy is glued into the match found by the kernel rule in the previous evaluation over $H_{[5]}$ and the duration of the marking node $a_{\exists_2}$ for that match is re-computed to $[5,7)$. The kernel rule finds also a new match containing `s` and `pm2`. The duration of the marking node $a'_{\exists_2}$ created for this match is empty, as no corresponding marking node created by $r_{n_{1.2}}$, and hence no multi-rule copy, exists.*

$r_{\cup}$ *A copy of the multi-rule is created for the new marking node created by $r_{\neg}$. This copy is glued into a new match for the LHS of the kernel, the one containing `s`, `pm2`, and $a'_{\exists_2}$. As $a'_{\exists_2}.d = \emptyset$, the duration of the marking node created for this match is also empty. As the match containing `s`, `pm1`, and $a_{\exists_2}$ has been modified ($a_{\exists_2}.d$ has changed), the rule is reapplied for this match and the duration of the marking node created, i.e., the marking node $a_{\cup}$ from the previous evaluation, is recomputed to [-55,7).*

r_{n_1} *Two matches are found. The rule is reapplied for both matches: the match m_1 (from before—containing `s` and `pm1`) and $a_{\cup}$ has been modified; the other match, containing `s` and `pm2` (or m_2) and the newly created marking node from the dependency, is new. The duration of the match involving m_1 is re-computed to $[4,7)$, whereas the duration of the match involving m_2 is empty.*

The evaluation returns a $\mathcal{G}^{\mathcal{V}}$ which has been incrementally updated and contains the match m_1. The match m_2, although structurally present, does not satisfy the MTGC in ζ_1—the match has an empty temporal validity. The temporal validity of m_1 has been updated to reflect the change to `d1` at time point 7.

4.2.3 *Preserving a High Degree of Incrementality*

As discussed in Section 3.3, an RTM$^{\text{H}}$ $H_{[\tau]}$ makes an assumption about the future of entities, i.e., the period (τ, ∞). Because of this assumption, the satisfaction computation $\mathcal{Z}$ may contain time points for which the satisfaction decision may be non-definite. Definite computations ($\mathcal{Z}^d$ and F) contain only time points for which a definite decision can be made. However, as the definite computations involve the time point of the RTM$^{\text{H}}$, every new instance of an RTM$^{\text{H}}$ calls for a re-computation of all definite spans; therefore, although matches continue to be incrementally maintained in the TVGDN, the computation of the temporal validity and temporal invalidity of matches is rendered non-incremental, thus reducing the degree of incrementality of the query evaluation. In addition, the query evaluation speed becomes dependent on the number of matches stored by the TVGDN. Since scalability is key to querying history at runtime, the default answer set featured in InTempo, as presented and implemented by this thesis, is $\mathcal{T}$, i.e., the answer set based on $\mathcal{Z}$, which preserves a high degree of incrementality in the query evaluation.

In the context of runtime monitoring, a property is typically monitored to detect falsifications—see Section 3.3.1. When used for runtime monitoring, $\mathcal{T}$ may include answers which, depending on the future of the execution, may become false positives. As an example, recall the query $\zeta_1 := (n_1, \psi_1)$ from Example 4.2.1 and Example 4.2.2, with $\psi_1 := \neg\exists\, n_{1.1}\, \mathrm{U}_{[0,60]}\, \exists\, n_{1.2}$. In order for matches of ζ_1 to return answers which constitute falsifications, i.e., there are admitted patients that are not prepared for treatment within the designated time or, until they are prepared, they are mistakenly re-triaged, the MTGC of the query needs to be negated: $\zeta_1' := (n_1, \neg\psi_1)$. Evaluated over $H_{[7]}$ in Figure 4.4, ζ_1' returns one match involving `pm2` and paired with the temporal validity $\mathcal{V} = [7, \infty)$, i.e., from time point 7 onward, the procedure for the patient with $pID = 2$ does not conform to the guideline.

The answer set over $H_{[7]}$ contains a match for `pm2`, i.e., a falsification, although, based on the interval of $\mathrm{U}_{[0,60]}$, the object may potentially satisfy ψ_1 in the future—for example, an addition of an instance of `DrugService` with the appropriate *pID* could occur in the next few seconds. If indeed this is the case, then the answer returned at time point 7 will have become a false positive.

This issue affects a class of properties C comprising MTGCs whose non-definiteness window w (see Section 3.3) is not zero, i.e., properties with future temporal operators, for which a decision at time point τ may depend on a future time point τ'. For properties outside C, a decision for τ can be made at τ and therefore decisions within the period that has been observed are definite. In runtime monitoring, properties in C constitute the most challenging cases as *the satisfaction decision at τ may have to be postponed until τ'*; thus, the running example used in the thesis, i.e., ψ_1, belongs to this class. The experimental evaluation presented in Chapter 5 contains properties in C that resemble ψ_1 as well as properties outside C. Technically, a reformulation of a $\psi \in C$ into a $\psi' \notin C$ may be possible. However, as also mentioned in Section 3.3.2, such a reformulation may be non-trivial.

In this section we equip Evaluation in InTempo with a technique to obtain an *effective answer set* $\mathcal{T}^e$ that, although based on $\mathcal{T}$, can be used for the detection of definite falsifications of properties in C that resemble ψ_1. The effective answer set $\mathcal{T}^e$ relies on the non-definiteness window w: it excludes a period based on w from the temporal validity of a match, effectively postponing returning a decision about time points in this period until the decision is definite—see also Corollary 3.3.1. We define the effective answer set $\mathcal{T}^e$ below, where we assume that w has been computed according to Definition 3.3.2.

Definition 4.2.1 (effective answer set $\mathcal{T}^e$). Let $\zeta := (n, \psi)$ be a query in $\mathcal{L}_{\mathrm{T}}$ and w be the non-definiteness window of ψ. Moreover, let $h^H_{\tau_{\mathcal{D}}}$ be a sequence of RTM$^{\mathrm{H}}$ instances, with $\mathcal{D} \in \mathbb{N}^+$. Given an answer set $\mathcal{T}(H_{[\tau_i]})$ for ζ over the RTM$^{\mathrm{H}}$

$H_{[\tau_i]}$ with $i \in [1, \mathcal{D}]$, the *effective answer set* $\mathcal{T}^e(H_{[\tau_i]})$ for ζ over $H_{[\tau_i]}$ is equal to $\mathcal{T}(H_{[\tau_i]})$ if $i = \mathcal{D}$; otherwise, $\mathcal{T}^e(H_{[\tau_i]})$ is the set of all tuples $(m, \mathcal{V} \cap [0, \tau_i - w])$ such that (i) $(m, \mathcal{V}(m, \psi)) \in \mathcal{T}(H_{[\tau_i]})$ and (ii) $\mathcal{V}(m, \psi) \cap [0, \tau_i - w] \neq \varnothing$.

The effective answer set $\mathcal{T}^e$ is intended for histories whose duration is larger than the non-definiteness window, i.e., $\tau_{\mathcal{D}} > w$. Moreover, the use of $\mathcal{T}^e$ relies on the MTGC containing no unbounded future operators—see Section 3.3.2.

The following theorem states that over a sequence of RTM$^{\mathrm{H}}$ instances with more than one member, for intermediate instances whose $\tau_i \geq w$, the $\mathcal{T}^e(H_{[\tau_i]})$ is equal to the *restricted definite temporal validity answer set* $\mathcal{T}^d_{\mathcal{V},r}(H_{[\tau_i]})$; this answer set is obtained based on the definite answer set $\mathcal{T}^d$ (see Section 3.3.4) but (i) only includes pairs of matches with their (non-empty) temporal validity $\mathcal{V}^d$, i.e., their temporal invalidity is omitted, and (ii) $\mathcal{V}^d$ is restricted to the period $[0, \tau_i - w]$.

Theorem 4.2.1 (equality of effective answer set and restricted definite temporal validity answer set over sequence of RTM$^{\mathrm{H}}$ instances)**.** *Let $\zeta := (n, \psi)$ be a query in $\mathcal{L}_{\mathrm{T}}$, w be the non-definiteness window of ψ, and $h^H_{\tau_{\mathcal{D}}}$ be a sequence of RTM^H instances with $\mathcal{D} \in [2, \infty] \cap \mathbb{N}^+$, and i be an index in $[k, \mathcal{D} - 1] \cap \mathbb{N}^+$ such that $\tau_k \geq w$. Moreover, let $\mathcal{T}^d_{\mathcal{V},r}(H_{[\tau_i]})$ be the restricted definite temporal validity answer set over $H_{[\tau_i]}$ which has been obtained from the definite answer set $\mathcal{T}^d$ but contains (i) only pairs of matches with their temporal validity $\mathcal{V}^d$ with $\mathcal{V}^d \neq \varnothing$ and (ii) $\mathcal{V}^d$ is intersected with $[0, \tau_i - w]$. Then, the effective answer set $\mathcal{T}^e(H_{[\tau_i]})$ is equal to $\mathcal{T}^d_{\mathcal{V},r}(H_{[\tau_i]})$, i.e.,*

$$\mathcal{T}^e(H_{[\tau_i]}) = \mathcal{T}^d_{\mathcal{V},r}(H_{[\tau_i]})$$

Proof. Based on the more general Theorem 3.3.2 which shows that, for $\tau \in [0, \tau_i - w]$, the satisfaction decision for τ in $H_{[\tau_i]}$ is equivalent to definite satisfaction decision for τ in $H_{[\tau_i]}$. The computations of $\mathcal{V}$ and $\mathcal{V}^d$ over $H_{[\tau_i]}$ rely on the computations of $\mathcal{Z}$ and $\mathcal{Z}^d$ over $H_{[\tau_i]}$, respectively. Theorem 3.2.1 and Theorem 3.3.3 show that satisfaction relation and definite satisfaction relation over $H_{[\tau_i]}$ are soundly reflected in $\mathcal{Z}$ and $\mathcal{Z}^d$ over $H_{[\tau_i]}$, respectively. □

As $\mathcal{T}^d_{\mathcal{V},r}$ excludes the definite falsification computation F, obtaining an effective answer set with F requires the evaluation of a separate query $\zeta' := (n, \neg\psi)$, i.e., a query searching for matches which falsify ψ, in parallel to ζ; this parallel evaluation would require the construction of a separate TVGDN for ζ'.

For an example of the effective answer set, consider the query ζ'_1 evaluated over $H_{[7]}$ mentioned above; the effective answer set $\mathcal{T}^e$ does not return the match involving `pm2`. Provided that no relevant change occurs that satisfies ψ_1 in the preceding period, $\mathcal{T}^e$ starts returning the match in query evaluations from time point 67 onward, i.e., when it has become a definite falsification.

If the MTGC ψ of a query is of the form $\neg\chi$, i.e., monitoring for falsifications, and not in C, then $\mathcal{T}^e$ returns answers timely; the temporal validity of matches may not include all definite time points, however all time points that are included in answers are definite. If the MTGC is monitoring for satisfactions, $\mathcal{T}^e$ introduces a delay in returning definite answers. For example, $\mathcal{T}^e$ for ζ_1 over $H_{[5]}$ does not contain the match involving `pm1` whose temporal validity is $[5, \infty)$; however, due to the definition of an RTM$^{\text{H}}$ which does not allow changes to creation timestamps, the match will exist at time point 5 and its temporal validity will include this time point for the rest of the execution; nevertheless, this match will start being included in $\mathcal{T}^e$ from time point 65 onward. For MTGCs that monitor for satisfactions, $\mathcal{T}$ may be more appropriate, as it will return matches timely, although their temporal validity may contain some time points for which the satisfaction decision may change. Alternatively, if used in monitoring for falsifications, $\mathcal{T}$ will return a *possible* falsification as early as possible, e.g., as is the case with the example based on $H_{[7]}$, and can be thus used as the basis for the generation of warnings.

The timing of changes may incur further delay. For example, in the evaluation of the query ζ_1' above, if a change does not occur at time point 67 but at a later time point, then the falsification is returned with a delay—see also Example 3.3.1. This type of delay however is inherent in reactive monitoring and, in practice, it is often handled by an appropriately timed periodic event generated by the monitoring procedure. In the context of adaptation, a delay in issue detection is often acceptable or even anticipated by adaptation schemes, as they are often based on a time-triggered feedback loop [see 91, 131]. An adaptation scheme based on InTempo can schedule a query evaluation, i.e., issue detection, in the future based on the non-definiteness window; regardless of whether a change occurs, this scheduled evaluation can make sure that a falsification of properties that resemble ψ_1 is not delayed.

4.3 MAINTENANCE

As noted previously, an RTM$^{\text{H}}$ maintains two views on the state of the modeled architecture. Similar to traditional RTMs, one view is of the current state, as the RTM$^{\text{H}}$ contains all entities present in the system. The other view, which extends traditional RTMs, is that of the entire history of the state. The view of the entire history is enabled by (i) creation (*cts*) and deletion (*dts*) timestamps (ii) retaining entities which have been deleted from the modeled system, the deletion being instead reflected in the *dts* of the entities in question.

This wealth in insight comes at a cost, which in certain cases might be problematic. On the one hand, regulations originating in the context of the system, such as retention policies for privacy-sensitive patient data in healthcare, e.g., [118, 119], may require that certain data has to be discarded after a certain period. On the other hand, remembering each system change causes the RTM$^{\text{H}}$ to constantly grow in size. The growth rate may need to be curbed to reduce the memory consumption or to avoid cluttering the model with obsolete data which may deteriorate performance of the pattern matching—as constantly more entities have to be considered.

For these cases, the remedy lies in knowing when and what to forget. In this section, we present an optional extension to InTempo, which is performed by Maintenance (see Figure 4.1) and allows for constraining the size of the history encoding. This constrained encoding contains entities that either have not yet been deleted in the modeled system or have been deleted but are relevant to query evaluations. Compared to the unconstrained representation, the constrained representation may afford increased memory efficiency.

4.3.1 *Discarding Irrelevant History from an RTM^H*

In case the *dts* of an entity is equal to ∞, the component modeled by the entity is still present in the architecture and thus, due to causal connection, may not be removed from the RTM$^{\mathrm{H}}$. This is not true for entities whose $dts \neq \infty$, as those have been removed from the modeled system. Nevertheless, because of the presence of temporal operators and their timing constraints, entities might be relevant to query evaluations for a certain period even after they have been deleted from the modeled system. When this period elapses, deleted entities may be considered for removal from the RTM$^{\mathrm{H}}$.

For example, let $\zeta_2 \coloneqq (n_{1.1}, \psi_3)$ be a query with $\psi_3 \coloneqq \lozenge_{[2,5]} \exists\, n_{1.2}$. Assume that ζ_2 is evaluated over $H_{[7]}$ in Figure 3.2. The entity `d1` has been deleted, yet it is relevant to the evaluation of the query. Given the timing constraint of the temporal operator in ζ_2, the same would stand for an evaluation over $H_{[10]}$: although deleted, `d1` might still be relevant to the query evaluation. The entity `d1` is no longer relevant at a time point τ, only for RTM$^{\mathrm{H}}$ instances whose associated timestamp exceeds the sum of the right end-point of the timing constraint of $\lozenge_{[2,5]}$ and τ. We define this time period of relevance as follows.

Definition 4.3.1 (relevance window $\mathcal{W}^r$)**.** Given a (finite) set of MTGCs Ψ, the *relevance window* $\mathcal{W}^r$, i.e., the period for which *deleted* entities may be relevant to query evaluations, is determined as follows. First, w^r for an MTGC $\psi \in \Psi$ is computed:

$$w^r(\psi) = \begin{cases} r(I) + \max(w^r(\chi), w^r(\omega)) & \text{if } \psi = \chi \mathrm{U}_I\, \omega \\ r(I) + \max(w^r(\chi), w^r(\omega)) & \text{if } \psi = \chi \mathrm{S}_I\, \omega \\ \max(w^r(\chi), w^r(\omega)) & \text{if } \psi = \chi \wedge \omega \\ w^r(\chi) & \text{if } \psi = \neg\chi \\ w^r(\chi) & \text{if } \psi = \exists(n, \chi) \\ 0 & \text{if } \psi = \mathit{true} \end{cases} \tag{4.1}$$

For Ψ, the relevance window is given by:

$$\mathcal{W}^r(\Psi) = \max_{\psi \in \Psi} w^r(\psi) \tag{4.2}$$

If the constrained representation option is enabled, $\mathcal{W}^r$ is computed prior to the evaluation of a query. Based on $\mathcal{W}^r$ and following Evaluation, Maintenance performs what resembles *garbage collection* and *prunes* every entity e in an RTM$^{\mathrm{H}}$ whose *dts* exceeds a certain threshold. This task yields a pruned RTM$^{\mathrm{H}}$, denoted by P and defined practically as follows.

$$P_{[\tau_i]} = \begin{cases} H_{[\tau_i]} & \text{if } i = 1 \\ \{e \mid e \in H_{[\tau_i]} \wedge e.dts \geq \tau_{i-1} - 2\mathcal{W}^r\} & \text{otherwise} \end{cases} \tag{4.3}$$

The pruning threshold $\tau_{i-1} - 2\mathcal{W}^r$ spans a period for which deleted entities can be relevant to the query evaluation at τ_i, i.e., $[\tau_{i-1} - \mathcal{W}^r, \tau_i]$. However, to ensure soundness, the threshold also covers a preceding period, i.e., $[\tau_{i-1} - 2\mathcal{W}^r, \tau_{i-1} - \mathcal{W}^r)$, which is motivated in the following section.

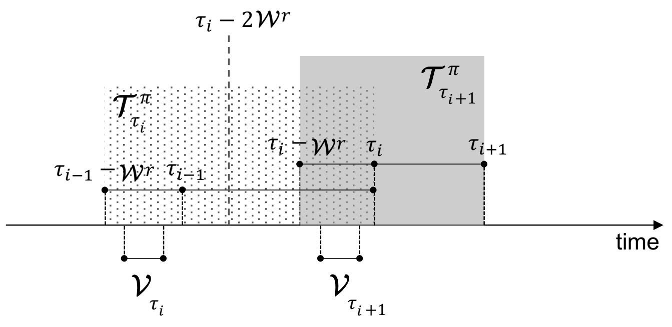

$\tau_i - 2\mathcal{W}^r$
$\mathcal{T}^\pi_{\tau_{i+1}}$
$\mathcal{T}^\pi_{\tau_i}$
$\tau_i - \mathcal{W}^r$
τ_i
τ_{i+1}
$\tau_{i-1} - \mathcal{W}^r$
τ_{i-1}
time
$\mathcal{V}_{\tau_i}$
$\mathcal{V}_{\tau_{i+1}}$

Figure 4.5: Projected answer sets $\mathcal{T}^\pi_{\tau_i}$ and $\mathcal{T}^\pi_{\tau_{i+1}}$ obtained for two consecutive time points τ_i and τ_{i+1}

4.3.2 *Projected Answer Set*

Given a history h_τ, i.e., a sequence of timed changes whose last member has the time point τ, an $H_{[\tau]}$ constitutes a *complete* encoding of all changes from the beginning of h_τ up to and including the time point τ. The removal of entities from $P_{[\tau]}$ renders this encoding *partial*. In the following, it is shown that all changes to the temporal validity $\mathcal{V}$ of a match can be detected both over a sequence of complete encodings and over a sequence of partial encodings. However, the $\mathcal{V}$ returned by partial encodings is restricted to a certain period, to correspond to the representation of the history duration being similarly confined. This restriction is captured by a *projected answer set* $\mathcal{T}^\pi$. Intuitively, a $\mathcal{T}^\pi$ projects the $\mathcal{V}$ of a match at τ_i, i.e., $\mathcal{V}_{\tau_i}$, on a period in which $\mathcal{V}_{\tau_i}$ might have changed (captured by $\mathcal{W}^r$) since the previous event at τ_{i-1}, that is, the interval $\Gamma = [\tau_{i-1} - \mathcal{W}^r, \tau_i]$.

Definition 4.3.2 (projected answer set $\mathcal{T}^\pi$)**.** Let ζ be a query from a set of queries Q in $\mathcal{L}_\mathrm{T}$ and $\mathcal{W}^r$ the relevance window of the MTGCs in Q. Moreover, let $\mathcal{T}_{\tau_i}(H_{[\tau_i]})$ be the answer set for ζ over the RTM$^\mathrm{H}$ $H_{[\tau_i]}$. Finally, let $\Gamma = [\tau_{i-1} - \mathcal{W}^r, \tau_i]$. Then, the *projected answer set* $\mathcal{T}^\pi_{\tau_i}(H_{[\tau_i]})$ is defined as follows.

$$\mathcal{T}^\pi_{\tau_i}(H_{[\tau_i]}) = \begin{cases} \mathcal{T}_{\tau_i}(H_{[\tau_i]}) & \text{if } i = 1 \\ \{(m, \mathcal{V} \cap \Gamma) \mid (m, \mathcal{V}) \in \mathcal{T}_{\tau_i}(H_{[\tau_i]})\} & \text{otherwise} \end{cases} \tag{4.4}$$

$\mathcal{T}^\pi$ is defined identically for a pruned RTM$^\mathrm{H}$ $P_{[\tau_i]}$.

For a match $(m, \mathcal{V})$, all past changes that could have affected its $\mathcal{V}_{\tau_i}$ are within $[\tau_i - \mathcal{W}^r, \tau_i]$. Let $\mathcal{T}^\pi_{\tau_{i+1}}$ be the answer set at the time point of the change after τ_i; $\mathcal{T}^\pi_{\tau_{i+1}}$ covers the interval $[\tau_i - \mathcal{W}^r, \tau_i]$ as well as the interval $[\tau_i, \tau_{i+1}]$. Therefore, with respect to τ_{i+1}, all time points in $\mathcal{V}_{\tau_{i+1}}$ before $\tau_i - \mathcal{W}^r$ are definite, i.e., no change after τ_{i+1} could affect them. This fact also motivates the pruning threshold: A deleted entity is kept in P as long as its *dts* is larger or equal to the earliest time point for which its existence could affect the earliest time point for which $\mathcal{V}$ is non-definite.

See Figure 4.5 for an illustration. The $\mathcal{T}^\pi_{\tau_i}$ (the dotted grid) returns a tuple $(m, \mathcal{V}_{\tau_i})$ where $\mathcal{V}_{\tau_i}$ contains all those time points which are definite with respect to τ_i. Analogously, so does $\mathcal{T}^\pi_{\tau_{i+1}}$ (the gray grid) with $\mathcal{V}_{\tau_{i+1}}$. At τ_{i+1} however, all time points before the pruning threshold $\tau_i - 2\mathcal{W}^r$ (that were non-definite in $\mathcal{T}^{\pi}_{\tau_i}$) *have become definite.*

The following theorem states that an aggregation of the projected answer sets $\mathcal{T}^{\pi}_{\tau_i}$ and $\mathcal{T}^{\pi}_{\tau_{i+1}}$ would still contain definite time points in $\mathcal{V}_{\tau_i}$ even if the pruning of an entity at τ_{i+1} would cause these time points to be excluded from $\mathcal{V}_{\tau_{i+1}}$.

Theorem 4.3.1 (equality of aggregation of projected answer sets over a sequence of pruned RTM$^{\mathrm{H}}$ instances and a sequence of complete RTM$^{\mathrm{H}}$ instances)**.** *Let $\zeta := (n, \psi)$ be a query in $\mathcal{L}_{\mathrm{T}}$, $h^H_{\tau_{\mathcal{D}}}$ be a sequence of complete RTM^H instances with $\mathcal{D} \in \mathbb{N}^+$, and $h^P_{\tau_{\mathcal{D}}}$ the corresponding sequences of pruned RTM^H instances. Then, the aggregation of the projected answer set $\mathcal{T}^{\pi}$ for ζ over $h^H_{\tau_{\mathcal{D}}}$ is equal to the aggregation of $\mathcal{T}^{\pi}$ for ζ over $h^P_{\tau_{\mathcal{D}}}$, that is:*

$$\bigcup_{i=1}^{\mathcal{D}} \mathcal{T}^{\pi}_{\tau_i}(H_{[\tau_i]}) = \bigcup_{i=1}^{\mathcal{D}} \mathcal{T}^{\pi}_{\tau_i}(P_{[\tau_i]}) \tag{4.5}$$

Proof (idea). By induction over $\mathcal{D}$. See Section A.3.8 for the proof. □

Since the history captured in P is partial, queries over P can compute the temporal validity of matches only for a restricted period of time—captured by the projected answer set. This is in contrast to queries over a complete RTM$^{\mathrm{H}}$ H where the temporal validity of matches is computed for the entire history. The loss of information in P compared to H is a trade-off for the potential of increased memory-efficiency and faster query evaluation times over P.

Moreover, the deletion of entities may result into matches being removed from P, therefore P is intended for use-cases where matches are only relevant for a short period of time after being returned, e.g., in self-adaptation, where a match constitutes an adaptation issue that is to be fixed as soon as possible.

As discussed in Section 4.2.3, for some properties the use of the effective answer $\mathcal{T}^e$ set from Definition 4.2.1 may be more appropriate. Therefore, we define a *projected effective answer set* $\mathcal{T}^{\pi,e}$ which, similarly to $\mathcal{T}^{\pi}$ and $\mathcal{T}$, restricts the temporal validity of $\mathcal{T}^e$ to a certain period of the history.

Definition 4.3.3 (projected effective answer set $\mathcal{T}^{\pi,e}$)**.** Let ζ be a query from a set of queries Q in $\mathcal{L}_{\mathrm{T}}$, $\mathcal{W}^r$ the relevance window of the MTGCs in Q, and w the non-definiteness window of the MTGC of ζ. Moreover, let $h^H_{\tau_{\mathcal{D}}}$ be a sequence of RTM$^{\mathrm{H}}$ instances with $\mathcal{D} \in \mathbb{N}^+$, and $\mathcal{T}^e_{\tau_i}(H_{[\tau_i]})$ be the effective answer set for ζ over an RTM$^{\mathrm{H}}$ $H_{[\tau_i]}$ with $i \in [k, \mathcal{D}-1]$ and $\tau_k \geq w$. Finally, let $\Gamma' = [\tau_{i-1} - \mathcal{W}^r, \tau_i - w]$. Then, the *projected effective answer set* $\mathcal{T}^{\pi,e}_{\tau_i}(H_{[\tau_i]})$ is defined as follows.

$$\mathcal{T}^{\pi,e}_{\tau_i}(H_{[\tau_i]}) = \begin{cases} \mathcal{T}^e_{\tau_i}(H_{[\tau_i]}) & \text{if } i = k \\ \{(m, \mathcal{V} \cap \Gamma') \mid (m, \mathcal{V}) \in \mathcal{T}^e_{\tau_i}(H_{[\tau_i]})\} & \text{otherwise} \end{cases} \tag{4.6}$$

$\mathcal{T}^{\pi,e}$ is defined identically for a pruned RTM$^{\mathrm{H}}$ $P_{[\tau_i]}$.

Similarly to Theorem 4.3.1, the following lemma states that the aggregation of $\mathcal{T}^{\pi,e}$ over a sequence of pruned RTM$^{\mathrm{H}}$ instances and a sequence of complete RTM$^{\mathrm{H}}$ instances returns the same results.

Lemma 4.3.1 (Equality of aggregation of projected effective answer sets over a sequence of pruned RTM$^{\mathrm{H}}$ instances and a sequence of complete RTM$^{\mathrm{H}}$ instances)**.** *Let $\zeta \coloneqq (n, \psi)$ be a query in $\mathcal{L}_{\mathrm{T}}$, $h^{H}_{\tau_{\mathcal{D}}}$ be a sequence of complete RTM^{H} instances with $\mathcal{D} \in \mathbb{N}^{+}$, and $h^{P}_{\tau_{\mathcal{D}}}$ the corresponding sequences of pruned RTM^{H} instances. Then, the aggregation of the projected effective answer set $\mathcal{T}^{\pi,e}$ for ζ over $h^{H}_{\tau_{\mathcal{D}}}$ is equal to the aggregation of $\mathcal{T}^{\pi,e}$ for ζ over $h^{P}_{\tau_{\mathcal{D}}}$, that is:*

$$\bigcup_{i=1}^{\mathcal{D}} \mathcal{T}^{\pi,e}_{\tau_i}(H_{[\tau_i]}) = \bigcup_{i=1}^{\mathcal{D}} \mathcal{T}^{\pi,e}_{\tau_i}(P_{[\tau_i]}) \tag{4.7}$$

Proof. The proof proceeds similarly to Theorem 4.3.1, i.e., by induction over $\mathcal{D}$. Intuitively, this is justified by $\mathcal{T}^{e}$ being a restricted version of $\mathcal{T}$, while the restriction concerns a time period which is not affected by pruning. □

4.3.3 *Maintenance with Dynamic Sets of Queries*

InTempo supports dynamic sets of queries over a complete RTM$^{\mathrm{H}}$. The following functionality enables their support by a pruned RTM$^{\mathrm{H}}$.

If the option to perform maintenance is enabled, $\mathcal{W}^{r}$ has to be re-computed every time the set of queries has been altered. If the newly computed $\mathcal{W}^{r}$ hasn't changed or has been decreased, the query evaluation proceeds as usual. Else, if $\mathcal{W}^{r}$ has increased, the derivation of a sound answer set for the *newly introduced* queries cannot be ensured for a period of time that is equal to the difference of the previous value of $2\mathcal{W}^{r}$ to the newly computed one. In this case, the queries for which a sound answer set cannot be guaranteed, are not admitted for evaluation until the length of the history represented by the RTM$^{\mathrm{H}}$ suffices for soundness.

For example, assume an evaluation of the previously introduced query ζ_2 with $\Psi = \{\psi_3\}$ and $\mathcal{W}^{r} = 5$. At a later evaluation, the query ζ_1 is added to the input queries which leads to $\Psi = \{\psi_1, \psi_3\}$. The time point of addition is marked by τ of the RTM$^{\mathrm{H}}$ $H_{[\tau]}$. The alteration induces a re-computation of $\mathcal{W}^{r}$ to $\mathcal{W}'^{r} = 60$. The value has been increased therefore, based on τ as well as the difference $2\mathcal{W}'^{r} - 2\mathcal{W}^{r}$, ζ_1 is admitted for evaluation only when an RTM$^{\mathrm{H}}$ $H_{[\tau']}$ is induced with $\tau' - \tau \geq 2\mathcal{W}'^{r} - 2\mathcal{W}^{r}$, i.e., in this case, at least 110 time units after its addition.

4.3.4 *Considerations*

In most cases, a pruned RTM$^{\mathrm{H}}$ would be more memory-efficient than a complete one. In fact, against a constant rate of incoming events, pruning would yield an RTM$^{\mathrm{H}}$ whose memory consumption would be bounded. However, if the event rate does not have a fixed upper bound, the memory consumption, although mitigated by pruning, would still increase over time.

In case an MTGC contains an unbounded past operator, i.e., with $r(I) = \infty$, an RTM$^{\mathrm{H}}$ is not pruned as the temporal requirement refers to the entire history.

Pruning may reduce the size of the matching search space and thus may improve the pattern-matching time during the query evaluation. On the other hand, pruning requires the usage of additional resources, which should be taken into consideration with respect to the overall evaluation performance.

First, pruning requires the storage and update of a priority queue of deleted entities in the RTM^{H}; the queue should be iteratively polled such that entities whose *dts* exceeded the threshold can be detected and pruned. Second, every time an entity is pruned from the RTM^{H}, queries have to be reevaluated based on the pruned RTM^{H}.

4.4 APPLICATION SCENARIO: HISTORY-AWARE SELF-ADAPTATION

A *Self-adaptive System* (SAS) is able to modify its own behavior or structure in response to its perception of its context, itself, and its requirements [43]. Self-adaptation can be generally achieved by adding, removing, and re-configuring components as well as connectors among components in the system architecture [105], therefore, the architecture view is typically considered an appropriate abstraction level [62]. Causal connection renders RTMs a natural choice for representing the architecture, as adaptations can be realized as changes on the RTM, which are subsequently mirrored in the system [see 158].

An established method of instrumenting self-adaptation is to equip the system with an external feedback control loop, i.e., an *adaptation engine*. A well-known reference model for the design of an adaptation engine is the *MAPE-K* feedback loop [93]. The MAPE-K loop $\underline{\text{m}}$onitors and $\underline{\text{a}}$nalyzes the system, and, if needed, $\underline{\text{p}}$lans and $\underline{\text{e}}$xecutes an adaptation of the system, where the adaptation is defined in terms of architecture changes. All four MAPE activities (whose first letter is underlined above) are based on $\underline{\text{k}}$nowledge. The feedback loop maintains an RTM as part of its knowledge to represent the current state of the architecture. Thus, the activities of the MAPE-K feedback loop operate on the RTM to perform self-adaptation.

An SAS can be adapted via *adaptation rules*. Adaptation rules represent fine-grained units of change that can be performed on the underlying system. The application of an adaptation rule adapts the system from its current state to a new state. Adaptation rules are of the form "*if* condition *then* action". A condition checks whether an *adaptation issue* is present, whereas an action describes a desired adaptation. If the condition is met, the action is taken. The feedback loop captures changes (during Monitor); checks whether changes cause an adaptation issue (during Analyze); and, if the condition is satisfied, plans and executes an adaptation action (during Plan and Execute, respectively) [101].

The graph-based encoding of architectural RTMs allows for a realization of adaptation rules in form of GT rules where adaptation issues are expressed via patterns which, in turn, characterize graph queries, i.e., the LHS of the rule; graph queries are evaluated during analysis and the RTM is adapted via in-place GTs based on the RHS of the rule [63].

An RTM^{H} captures the current state of an RTM as well as the history of the state. Queries in $\mathcal{L}_{T}$ are capable of formulating conditions, i.e., the LHS, in adaptation rules which include temporal requirements on the history of evolution of the pattern. By replacing the RTM in a MAPE-K loop with an RTM^{H}, the knowledge of the feedback loop is extended to cover history-awareness; based on the RTM^{H}, InTempo can be used to evaluate temporal queries which, in this context, correspond to checking adaptation conditions with temporal requirements, over a sequence of changes to the architecture. Query answers

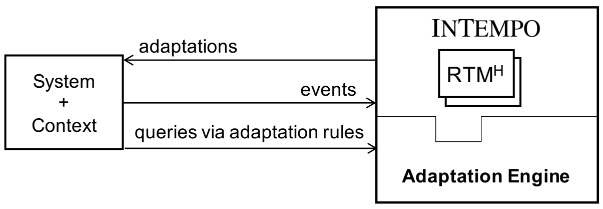


Figure 4.6: Overview of InTempo and system interaction for adaptation

can be used to plan adaptations. Therefore, InTempo may serve as the basis for an adaptation engine that enables *history-aware self-adaptation*—see Figure 4.6.

Figure 4.7 depicts a detailed view of a history-aware adaptation engine based on the MAPE loop. The engine operates in two phases: the *setup* and the (self-adaptation) *loop*. Operationalization of InTempo (the trapeze shape containing "O" in Figure 4.7) is performed during setup. Evaluation ("E") is performed in the Analyze activity. The engine contains a *Maintain* activity—an extension compared to a canonical MAPE loop—during which Maintenance of InTempo ("M") is performed.

4.4.1 *Self-adaptation Scenario for SHS*

In the following, we build on the SHS introduced in Example 2.1.1 to envisage a (self-)adaptation scenario that enacts a medical instruction. The instruction imposes temporal requirements on the operation of the SHS which are checked and enforced by the five activities of the adaptation loop described below. The scenario is based on the medical guideline on the treatment of sepsis [108, 126], a potentially life-threatening condition. We focus on the basic instruction that reads: *"between ER Sepsis Triage and IV Antibiotics should be less than 1 hour"* [108]. Note that, from the viewpoint of the system, this timing constraint in the instruction is soft, as medical guidelines often provide contingency plans in case a deadline is inadvertently missed, e.g., [162, p. 11]. Therefore, a system adaptation could occur after the deadline is missed and still remedy the situation.

The event log in [108] contains real medical records from a hospital where patients diagnosed with sepsis were treated according to the guideline. The log

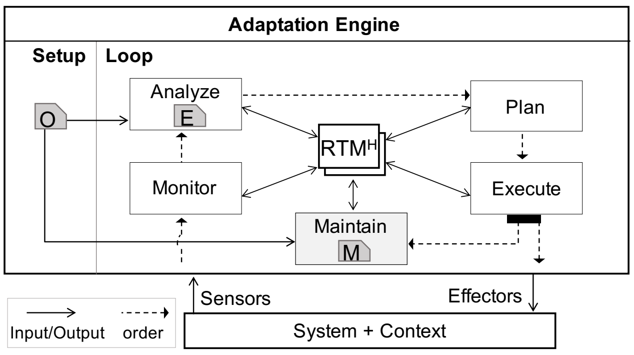


Figure 4.7: History-aware adaptation engine

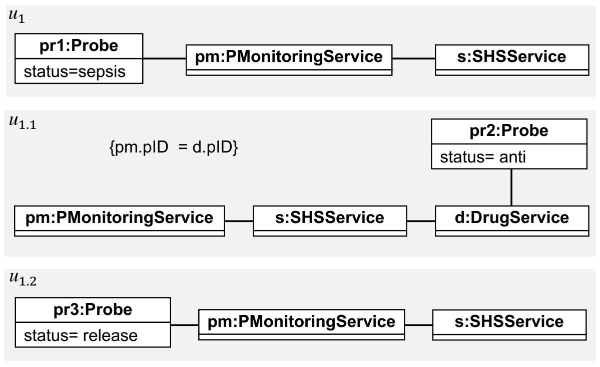


Figure 4.8: Patterns for self-adaptation in SHS—the usage of the same label for vertices denotes that vertices refer to the same entity in the RTM^{H}

contains a multitude of events, we, however, focus on those that correspond to actions prescribed in the guideline, i.e., *ER Sepsis Triage*, *IV Antibiotics*, and *Release*, which correspond to a patient being triaged as an emergency sepsis case, an intravenous (IV) administration of antibiotics for sepsis, and a patient being released from the emergency ward, respectively. Events in the log are timestamped. Based on the SHS metamodel (Figure 2.1) and the available hospital log, we envisage the procedure described in the guideline enacted by the SHS.

In detail, an *ER Sepsis Triage* event is simulated as a `Probe` with `status` *sepsis*, generated from an instance of `PMonitoringService` `pm` which has been invoked by an `SHSService` `s`. An *IV Antibiotics* event is simulated as a `Probe` with `status` *anti* from a `DrugService` `d` which has also been invoked by `s`. The patterns capturing the occurrence of these events in our SHS are depicted in Figure 4.8. To make sure these two actions are referring to the same patient, the pattern $u_{1.1}$ contains an attribute constraint (in braces) that checks whether the *pID* of `d` and `pm` are equal.

Based on u_1 and $u_{1.1}$ in Figure 4.8, the instruction is formulated in $\mathcal{L}_T$ by the query MG1 $:= (u_1, \neg\psi_{MG1})$ which searches for falsifications of the MTGC $\psi_{MG1} := \Diamond_{[0,3600]}\exists u_{1.1}$. That is, the query searches for matches of u_1, which identifies a (previously untreated) patient with sepsis, for whom, in the next hour, there is no match for pattern $u_{1.1}$, which identifies the administration of antibiotics to the *same* patient. The binding of elements in $u_{1.1}$ from u_1 is illustrated in Figure 4.8 by using the same labels for vertices, e.g., for `pm`. The system is assumed to track time in seconds.

In order to challenge our approach with a more complicated scenario, we also search for falsifications for a variation of MG1. Namely, that no patient with sepsis should be released prior to being treated, a requirement that resembles process compliance checks and refers to *Release* events in the original log. The requirement is captured by the query MG2 $:= (u_1, \neg\psi_{MG2})$ with $\psi_{MG2} := \neg\exists u_{1.2}\,\mathrm{U}_{[0,3600]}\exists u_{1.1}$. The pattern $u_{1.2}$ is similar to u_1 and depicted in Figure 4.8. We describe a desired adaptation loop for the SHS according to these instructions.

Monitor During the monitoring activity, the recent events (new readings captured by `Probes` since the last invocation of the loop) together with their *cts* and *dts* values are reflected in the RTM^{H}, which is an instantiation of the `Architecture`. Therefore, the RTM^{H} is updated to represent the current system structure extended with the relevant temporal data.

Analyze The activity detects adaptation issues. In this context, these are captured by falsifications of ψ_{MG1}, i.e., the existence in the RTM^{H} of structural patterns that reflect sepsis cases (u_1) without associated antibiotics ($u_{1.1}$) within one hour; or, for ψ_{MG2}, the existence of sepsis cases that are either not administered antibiotics within one hour or they are erroneously released ($u_{1.2}$) without first being timely treated. Hence, we evaluate (separately) the queries MG1 and MG2. The evaluation comprises the executions of the TVGDNs obtained by Operationalization during the setup of the engine. In the context of an SHS, self-adaptation only *supports* the medical procedure, i.e., it first waits for clinicians to perform the actions in the guideline. Only when there is no more time is self-adaptation enabled. The effective answer set (see Section 4.2.3) returned by InTempo fulfills this expectation, as it restricts the temporal validity of matches based on the non-definiteness window; thus, the period in which the decision for time points in this answer set could change has elapsed.

The matches returned by InTempo in this activity constitute adaptation issues, and similar to [63], adaptation-related types (`Annotation` in Figure 2.1) are used to facilitate the adaptation. During Analyze, the `PMonitoringService` instance which has been involved in the detection of an adaptation issue is annotated with an `Issue` instance. Therefore, to ensure that only new falsifications are matched, u_1 is extended to check that no instance of `Issue` is associated with the matched instance of `PMonitoringService`. `Issue` instances, as well as instances of other adaptation-related nodes in Figure 2.1, are created by appropriately typed GT rules which are capable of setting their *cts* and *dts*.

Plan and execute In planning, the engine searches for *sepsis* `Probe`s annotated with an instance of `Issue`. Upon finding them, it attaches an `Effector` to the service to which the `Probe` instance is attached. The Execute activity of the loop searches for effectors and upon finding them takes an adaptation action, i.e., administers antibiotics to the patient via a `DrugService` instance. This adaptation action is also reflected in the RTM^{H} by creating an `AdaptationAction` instance which is associated to the handled `Issue` instance.

Maintain This activity is optional. If enabled, Maintenance uses the relevance window obtained by Operationalization during setup and prunes the RTM^{H}, i.e., it removes all entities that have been deleted and are irrelevant to query evaluations. Following the removal of entities, the TVGDN is re-executed to update matches.

4.5 SUMMARY

This chapter presented InTempo, a querying approach for temporal queries in $\mathcal{L}_T$. InTempo is a collection of three interdependent operations: Operationalization, Evaluation, and Maintenance. Operationalization (Section 4.1) transforms a (declarative) temporal query into a network of sub-queries which is amenable to incremental evaluation (contribution C3.1 in Section 1.3). Evaluation (Section 4.2) performs the query evaluation. Maintenance (Section 4.3) is an optional operation that prunes deleted entities from the RTM^{H} that are not relevant to query evaluations, thus affording a memory-efficient history encoding (C3.2). Section 4.4 presents a typical application scenario which integrates these operations in an adaptation engine that is rendered capable of history-aware adaptations.

Operationalization relies on a novel formal framework, presented in Section 4.1, which decomposes a query in $\mathcal{L}_{\text{T}}$ into a directed acyclic graph consisting of sub-queries called TGDN. Each sub-query characterizes a GT rule, also called a marking rule, as the RHS of the rule creates a marking node that marks matches for the LHS of the rule. The marking node has a distinguished attribute called *duration* which stores the satisfaction span of the match. The framework captures the conditionals in the computations for operators in MTGL by Amalgamated Marking Rules. Given a query in $\mathcal{L}_{\text{T}}$, Operationalization constructs a network according to the framework whose result computes the answer set $\mathfrak{T}$ of the query. We summarize the introduced concepts below.

- *TGDN* (denoted by g): Given a query $(n, \psi) \in \mathcal{L}_{\text{T}}$, a TGDN is a GTS consisting of a set of basic and amalgamated marking rules derived by ψ.
- *Satisfaction span set* (Σ): A declarative definition of a set containing all matches for a pattern n in an RTM^{H} paired with their satisfaction span $\mathfrak{Z}$.
- *TGDN result* (G): A set containing matches marked by marking nodes created by the root of g paired with the duration of their marking nodes.
- *Terminal rule* (r_t): The terminal rule corresponds to the query pattern n and is dependent on the root of the TGDN g.
- *TVGDN* ($g^{\mathcal{V}}$): The network g following the addition of the terminal rule.
- *TVGDN result* ($G^{\mathcal{V}}$): A set containing matches marked by marking nodes created by the terminal rule paired with the duration of their marking nodes.

Given the query (n, ψ), Lemma 4.1.1 shows that, over a given RTM^{H}, the TGDN result G constructed by Operationalization is equal to the satisfaction span set Σ. Theorem 4.1.1 shows that the TVGDN result is equal to the answer set $\mathfrak{T}$ of the query. Section 4.1.6 presents the corresponding results for a construction of a network that computes the definite temporal validity and invalidity.

The TVGDN is amenable to incremental execution, thereby facilitating the incremental evaluation of the query. The incremental execution is performed by Evaluation in InTempo, as shown in Section 4.2. However, the degree of incrementality would be significantly reduced in the execution of a TVGDN for definite computations, as the execution would require that all satisfaction spans of matches are updated for every instance of the RTM^{H}.

Aiming to preserve a high degree of incrementality, InTempo is based on the construction of a TVGDN for (non-definite) satisfaction and introduces the *effective answer set* $\mathfrak{T}^e$ in Section 4.2.3. $\mathfrak{T}^e$ is based on the answer set $\mathfrak{T}$, thus it does not require the satisfaction span of matches to be updated for every instance of the RTM^{H}. On the other hand, for certain properties, $\mathcal{T}$ could include some false positives. To avoid this, $\mathcal{T}^{e}$ does not return results until they have become definite. Thus, $\mathcal{T}^{e}$ affords highly incremental evaluations and definite results, at the cost of delivering these results with a delay for certain properties. Theorem 4.2.1 shows that $\mathcal{T}^{e}$ is equal to a version of the definite answer set $\mathcal{T}^{d}$ (see Section 3.3.4) where the temporal validity is restricted based on the non-definiteness window.

Maintenance prunes deleted entities from the RTM^{H} that are not relevant to query evaluations and may thus reduce the size of the encoded history. Pruning relies on the *relevance window* ($\mathcal{W}^{r}$), which is computed based on the timing constraints of temporal operators in an MTGC. To correspond to the extent of history covered by a pruned RTM^{H}, we introduce the *projected answer set* $\mathcal{T}^{\pi}$ which restricts the temporal validity for matches based on $\mathcal{W}^{r}$. Theorem 4.3.1 shows that, given a sequence of pruned RTM^{H} instances a and a sequence of unpruned RTM^{H} instances b, the union of projected answer sets over a is equal to the union of projected answer sets over b.

Before concluding this chapter, we summarize the introduced answer sets for $\mathcal{L}_{T}$ and their characteristics in Table 4.5; we omit the projected effective answer $\mathcal{T}^{\pi,e}$ from Definition 4.3.3 which is a combination of $\mathcal{T}^{\pi}$ and $\mathcal{T}^{e}$.

Table 4.5: An overview of the introduced answer sets for $\mathcal{L}_{T}$ and their characteristics

Notation	Name	Reference	Characteristics
$\mathcal{T}$	-	Definition 3.2.3	highly incremental featured in InTempo may contain false positives
$\mathcal{T}^{d}$	definite	Definition 3.3.4	reduced incrementality not featured in InTempo definite answers
$\mathcal{T}^{e}$	effective	Definition 4.2.1	highly incremental implemented based on $\mathcal{T}$ definite but possibly delayed answers
$\mathcal{T}^{\pi}$	projected	Definition 4.3.2	highly incremental featured in InTempo temporal validity is restricted

5 EXPERIMENTAL EVALUATION

This chapter presents the experimental evaluation of InTempo. The experimental evaluation aims to measure the performance and assess the fulfillment of the objective for increased scalability (see Section 1.3). In this context, scalability refers to the capability of InTempo to mitigate the dependence of query evaluation times and memory consumption on the size of the history. The objective is assessed based on a comparison of the performance of InTempo to two state-of-the-art solutions from the RV and MDE communities. Performance is measured in two case-studies: a simulation of the history-aware self-adaptation scenario for the SHS introduced in Section 4.4, and a partial execution of the *Social Network Benchmark* (SNB) by the Linked Data Benchmark Council.

Section 5.1 presents a prototypical implementation of InTempo.

Section 5.2 presents the input data, queries, and experimental setting for the SHS. The SHS evaluation is based on both real and synthetic data; we have generated the synthetic data using sophisticated statistical methods, thereby enabling the measurement of performance of InTempo over increasing history sizes which preserve the statistical characteristics of the real log.

Section 5.3 presents the input data, queries, and experimental setting for the SNB. The SNB log files allow for a more extensive experimental evaluation of InTempo as they involve more complex, realistic, and considerably larger graph structures. $\mathcal{L}_T$ is used for more complex queries than those in the SHS case-study which stem from a domain which is significantly different to healthcare.

In Section 5.4, we conduct the experiments with two state-of-the-art tools from the RV and MDE communities and measure their performance.

Section 5.5 discusses the results and the extent to which they fulfill the objective for increased scalability. Moreover, following the experimental evaluation, the section reflects on the advantages and limitations of InTempo compared to the state-of-the-art. Finally, this section discusses the threats to the validity of the experimental evaluation.

5.1 IMPLEMENTATION

Our implementation of InTempo is based on the *Eclipse Modeling Framework* (EMF) [51, 149], which is a widespread MDE technology for creating software systems. For pattern matching, we employ the *Story Pattern Matcher* [66, 114] using the search plan generation strategy presented in [5]. The Matcher uses local search (see Section 2.2.5) to start the search from a specific element of the graph and thus reduces the pattern matching effort [88]. It uses an *Object Constraint Language* (OCL) [127] checker for checking attribute constraints. For interval computations, we use an open-source library [72]. For the removal of elements from the RTM^H, we transparently replace the native EMF method, via a JAVA agent, with an optimized version which reduces the potentially expensive shifting of cells in the underlying array list and renders the removal more efficient. The implementation is available as an EMF plugin in [113]. For

```
import 'http://mdelab.de/intempo/examples/shs/1.0'

($u1, !(true U[0,3600] E $u11))

declarations{
    u1{ pr1:Probe
        pm:PMonitoringService
        s:SHSService
        pm -probes-> pr1
        s -connected-> pm
        [OCL:"pr1.status='sepsis'"]}
    ...
}
```

Listing 5.1: The query MG1 in ITQL (excerpt from Listing B.2)

the experimental evaluation, we developed two variants based on the plugin: **IT**, with pruning disabled, and **IT**$_{+P}$, with pruning enabled.

The plugin, henceforth simply called InTempo, can be used either via the EMF user interface or via an API. The API enables the utilization of InTempo answers in other applications, e.g., an adaptation engine. InTempo offers two operation modes intended for different scenarios: one mode supports the evaluation of a temporal query over a user-provided RTM^H, whereas the other assumes that, instead of being captured by an RTM^H, data about the system execution have been captured in an event log; in this mode, InTempo supports the mapping of log entries to RTM^H modifications based on a metamodel and an event mapping (see design-time artifacts in Section 4.1.1), and the subsequent query evaluation following each modification. More information on the operation modes of the plugin are presented in Section B.1.

InTempo supports the evaluation of queries in $\mathcal{L}_T$. To facilitate the specification of such queries, we introduce the *InTempo Query Language* (ITQL). ITQL supports the definition of Story Patterns, i.e., structural patterns that should be matched by the Matcher, via *declarations*. Declarations aim at making the specification of queries more concise and are used for building Story Patterns during the loading of an ITQL file. Listing 5.1 shows an excerpt of the specification of the query MG1 from Section 4.4.1 in ITQL; the specification relies on the declaration for the pattern u_1 from Figure 4.8 and the metamodel in Figure 2.1. The query syntax in ITQL complies to the syntax of $\mathcal{L}_T$. Thus, following u_1, is the AC of the query, i.e., the negation of $\psi_{MG1} := \Diamond_{[0,3600]} \exists u_{1.1}$, formulated in ITQL without the *eventually* abbreviation. To facilitate the formulation of intervals, ITQL support shorthand notations for referring to days, months, and years, as well as for specifying open end-points of intervals. The grammar of ITQL is shown in Section B.1.1.

For the mapping and translation of log entries into model modifications, we introduce the *Events-to-Patterns Specification Language* (E2P). An E2P specification consists of mappings between events, i.e., lines of comma-separated attribute-value pairs, and actions that should be performed on the RTM^H. E2P supports five actions (formulated as verbs): *adds*, to add an entity and optionally assign values to the attributes of the added entity; *adds-ref*, to add a connector between two entities; *modifies*, to modify the attribute values of an entity; *deletes* and *deletes-ref*, to delete an entity and a connector, respectively, from the RTM^H. To accommodate linked data, E2P allows for the indexation of added entities

```
import 'http://mdelab.de/intempo/examples/shs/1.0'

...

"ER Sepsis Triage":{adds pm:PMonitoringService >> PMServices(*p1) [ID=*p1]
    adds pr:Probe >> SProbes(*p1) [ID=*p1 status="sepsis" cts=*p3]
    adds-ref pm -probes-> pr
    adds-ref hospital -ownedServices-> pm}

...
```

Listing 5.2: An E2P mapping for the *ER Sepsis Triage* event in the SHS case-study

so that later events can refer to modifications that have been processed earlier. Listing 5.2 shows an example of an E2P mapping; the mapping describes the actions that should be performed on the RTM^H based on the event *ER Sepsis Triage* in the SHS case-study—see Section 4.4.1. The mapping relies on the SHS metamodel in Figure 2.1. We briefly explain an *adds* action: the event *ER Sepsis Triage* entails the addition of an entity `pm` of type `PMonitoringService`; this entity is added to an index called *PMServices* at a position that is equal to the value of the attribute-value pair at the first position of the log entry; the attribute *ID* of the created entity gets the same value. Indexing the created entity `pm` allows for later modifications to refer to `pm`, e.g., the *adds-ref* action that creates a connector between `pm` and an entity of type `Probe` that was created after `pm`. The grammar of E2P and more information on the capabilities of the language are presented in Section B.1.2.

5.2 TIMELY SEPSIS TREATMENT FOR THE SMART HOSPITAL SYSTEM

This section presents an implementation of InTempo integrated in an adaptation engine as shown in Figure 4.7. The engine is evaluated for an adaptation scenario which encompasses the monitoring of a temporal property at runtime. We developed a simulator of the adaptable SHS presented in Section 4.4.1. The simulation replays events on an RTM^H based on the real and synthesized event logs described in Section 5.2.1. After each event, the temporal queries MG1 and MG2, described in Section 4.4.1, are evaluated. Matches constitute adaptation issues which are resolved by appropriate modifications to the RTM^H.

5.2.1 *Input Logs*

The log used in our experiments (in the following, *real* log) contains 1049 *trajectories* of sepsis patients admitted to a hospital within 1.5 years [108]. Each trajectory comprises a sequence of events. The events that are relevant to the experiment are *ER Sepsis Triage* (ER), *IV Antibiotics* (IV), and *Release* (RE) events. A trajectory starts with an ER event, and IV and RE events might follow. The *inter-arrival time* (IAT) between two ER events defines the arrival rate of trajectories. We used statistical probability distribution fitting to find the best-fitting distribution that characterizes the inter-arrival times between: two ER events (IAT_T), an ER and an IV (IAT_{SA}), and an ER and an RE (IAT_{SR}). Then, we used statistical bootstrapping [44] to generate two synthetic logs, x10

Table 5.1: Overview of input logs: the number of events is mapped to the number of vertices and edges created in the model; the column Deleted shows the percentage of deletions for vertices and edges contained in the logs

Log	Events (#)	Vertices (#) / Edges (#)	Deleted (%)	Used in
real	8K	5K / 5K	100 / 100	
x10	88K	58K / 58K	100 / 100	
x100	874K	583K / 583K	100 / 100	SHS
M-real	21K	-	100 / 100	
M-x10	234K	-	100 / 100	
M-x100	2.3M	-	100 / 100	
sf-0.1	952K	900K / 3.4M	5.2 / 4.4	SNB
sf-1	10M	10.2M / 38.4M	4.3 / 3.4	

and x100, with IAT_T values that are 10 and 100 times smaller, respectively, than IAT_T values of the *real* log. IAT_{SA} and IAT_{SR} remain as in the *real* log.

As a result, x10 and x100 cover the same period of time as the *real* log, and increase the trajectory density (approx.) ten and a hundred times, respectively, allowing us to test the scalability of InTempo without altering the statistical characteristics of the *real* log. The logs are available in [133] and a more detailed description of the statistical methods employed is presented in Section B.2.1.

Each event in the logs corresponds to the creation of certain elements in the RTM^{H}. In order however to evaluate the performance of pruning we required that the lifespans of these elements have an end, i.e., their *dts* is set. This information is not provided in the original log. Therefore, for each created element we defined an interval after which a delete event was injected in the logs. The intervals for `Probe` and `Service` instances are ten seconds and one hour, respectively. The logs that contain the deletions are available in [134]. An overview of the logs is shown in Table 5.1, where values have been rounded. As shown by the *Deleted* column, all created elements are eventually deleted.

5.2.2 *Experiment Design*

We integrated both implementation variants, i.e., with and without pruning, in an adaptation engine. We denote this integration by an arrow circle: $\mathbf{IT}^{\circlearrowleft}$, includes the Monitor, Analyze, Plan, and Execute activity, i.e., in terms of InTempo, the operations Operationalization and Evaluation; $\mathbf{IT}^{\circlearrowleft}_{+\mathrm{P}}$, includes all activities above plus Maintenance, i.e., Operationalization, Evaluation, and Maintenance of InTempo. See Figure 4.7 for an overview. The experiments[1] simulate the events in the *real*, x10, and x100. Each experiment entails the execution of one variant for one query, either MG1 or MG2; the specification of the queries in ITQL are shown in Listing B.2 and Listing B.3, respectively.

1 All experiments have been conducted on an Intel E5-2643 with 256 GBs of DDR4 RAM and an OpenJDK8 JVM.

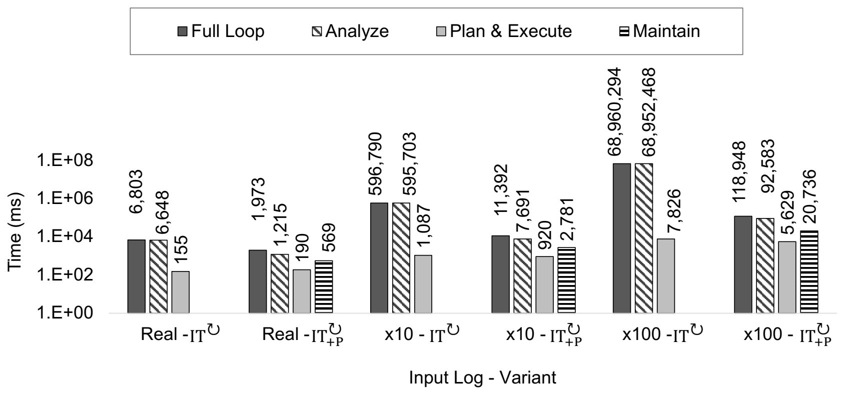


Figure 5.1: Cumulative time of loop activities for MG1

We measure $\mathrm{IT}^{\circlearrowright}$ and $\mathrm{IT}^{\circlearrowright}_{+\mathrm{P}}$ with respect to their *reaction time* (or full loop time). In this context, the reaction time is equal to the required time for one execution of the adaptation loop, i.e., the time from when an issue is detected to when a corresponding adaptation action has been performed. Thus, the reaction time consists of times for Analyze, Plan, Execute and, for $\mathrm{IT}^{\circlearrowright}_{+\mathrm{P}}$, Maintain. The time spent in Monitor, i.e., processing an event, is negligible and thus not reported. The time required for Analyze is the query evaluation time. Time measurements are used to assess the scalability of query evaluations times.

In each adaptation loop, we measure the memory consumed by the variants based on the values reported by the JVM. Memory consumption is used to assess the scalability of memory consumption.

A loop is invoked periodically based on a predefined but modifiable frequency. In our experiments, based on the IAT_T of the logs, we set the invocation frequency to one hour, i.e., 3600 seconds, to avoid frequent invocations where no events are processed. The invocation frequency coincides with the maximum delay of a falsification detection, i.e., in the worst case, a falsification will occur at the first second after the loop and will be detected at the next invocation which in this case is after almost one hour. Operationalization, which produces the TVGDN and the relevance window utilized by Maintenance of $\mathrm{IT}^{\circlearrowright}_{+\mathrm{P}}$, is performed only once during the setup of the loop.

Each experiment is measured for either time or memory and proceeds as follows. First, during Monitor activity, events from the logs are processed. Each log event corresponds to certain modifications to the $\mathrm{RTM}^{\mathrm{H}}$: an *ER Sepsis Triage* event corresponds to the addition of a `PMonitoringService` and a `Probe` instance to the model, where the attribute `status` of the `Probe` instance is set to *sepsis*; an *IV Antibiotics* event corresponds to the addition of a `DrugService` instance and a `Probe` instance with `status` *anti*; a *Release* event is similar to the *Er Sepsis Triage*, except the `status` is set to *release*. All added elements are eventually deleted by corresponding deletion events. All mappings are defined in the E2P specification shown in Listing B.1. The loop is invoked at the predefined intervals, triggering the Analyze activity which executes the query. Matches constitute adaptation issues. During Plan and Execute transformations are performed which correspond to adaptation actions. Finally, for $\mathrm{IT}^{\circlearrowright}_{+\mathrm{P}}$, Maintain is performed and matches are recomputed.

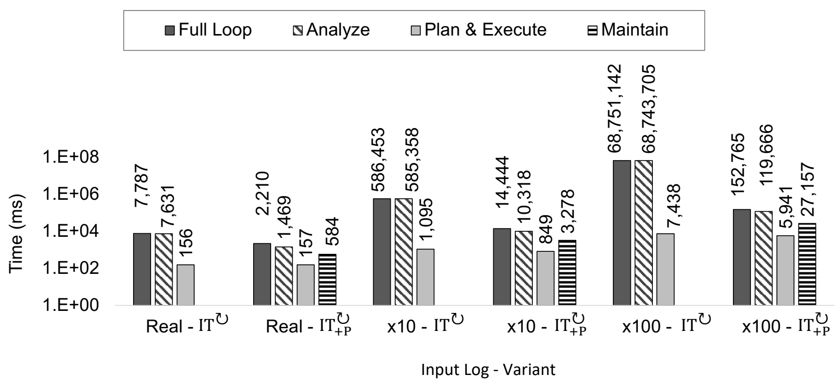


Figure 5.2: Cumulative time of loop activities for MG2

5.2.3 *Results*

Figure 5.1 and Figure 5.2 depict the cumulative time (in logarithmic scale) for each of the measured loop activities and the reaction time, i.e., full loop time, for MG1 and MG2.

As expected, the results are mainly influenced by the Analyze activity, which is when issues are detected, i.e., queries are evaluated. The number of processed events in the experiments with the log files *real*, x10, and x100 grows steadily—see Section 5.2.1. For $IT^{\circlearrowright}$, this increase is mirrored in the size of the RTM^{H}. The Analyze time of $IT^{\circlearrowright}$ increases with respect to these two parameters. The growth of the RTM^{H} can also be seen in Table 5.5, where the maximum memory measurement is reported for both variants. Contrary to $IT^{\circlearrowright}$, the pruning in $IT^{\circlearrowright}_{+P}$ minimizes the size and thereby the memory consumption of the RTM^{H}. In fact, because the rate of created elements per period in each log does not increase, and, over time, it is almost equal to the rate of deleted elements, the memory consumption over *real*, x10, and x100 remains unchanged.

Owing to pruning, the Analyze time of $IT^{\circlearrowright}_{+P}$ increases at a considerably smaller pace compared to $IT^{\circlearrowright}$. Pruning forces a re-computation of the answer set, therefore, as shown in Figure 5.1 and Figure 5.2, the time Maintain requires is non-negligible. Figure 5.3 shows the time spent in Analyze for each loop of the two variants for the x100 log (in logarithmic scale). The pruning of RTM^{H} enables the Analyze time of $IT^{\circlearrowright}_{+P}$ to remain constant.

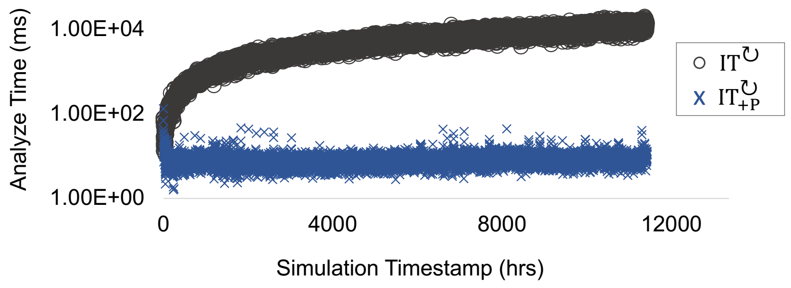


Figure 5.3: Time for Analyze activity per variant (MG1 - x100)

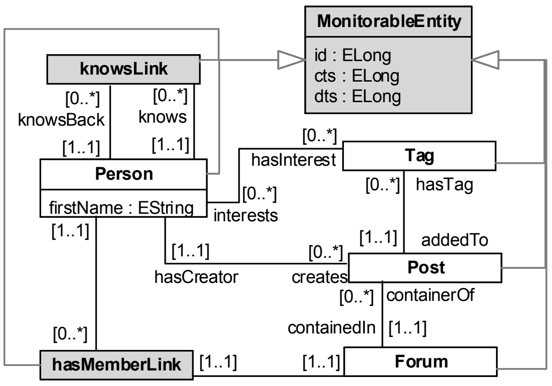


Figure 5.4: Relevant excerpt of the metamodel of the SNB

5.3 TREND DETECTION FOR THE SOCIAL NETWORK BENCHMARK

The *Social Network Benchmark* (SNB) from the Linked Data Benchmark Council [103] is designed to simulate a plausible social network in operation. The benchmark can generate data of varying sizes and provides a series of realistic usage scenarios which aim at stress-testing and discovering bottlenecks in graph-based technologies. Recent versions of the benchmark generate data which contain both *insert* and *delete* operations [160] and can be conveniently transformed into a stream of timestamped creation and deletion events.

Metamodel and Queries

The SNB metamodel consists of a *static* part, i.e., the entities `City`, `Country`, `Tag`, `TagClass`, `University`, and `Company` whose instances are created before the creation of the network and are never deleted, and a *dynamic* part, i.e., the entities `Person`, `Post`, `Comment`, and `Forum`, whose instances are created during the operation of the network and can be deleted. A relevant excerpt of the SNB metamodel is shown in Figure 5.4—where entities in gray are explained later in this section. In the generated data, forum memberships and friendships are represented by links between entities. The vast majority of the network activity comprises persons joining forums, befriending other persons, or posting comments and replies in forums.

From the available queries in the SNB specification, we select two queries with a temporal dimension, namely **IC4** and **IC5** [55]. The (slightly adjusted) query IC4 reads: "*Given a start* `Person`*, find* `Tag`*s that are attached to* `Post`*s that were created by that* `Person`*'s friends. Only include* `Tag`*s that were attached to friends'* `Post`*s created within a year after they became friends with the start* `Person`*, and that were never attached to friends'* `Post`*s created before that.*" Similarly to the SHS case-study, statements in the query are captured as patterns, shown in Figure 5.5.

The query refers to the point in time a friendship was created. Evaluating the query would entail checking the creation and deletion timestamp of the link that represents the friendship. InTempo does not directly support attributes in links. Following a customary modeling technique, e.g., [97], links of interest can be encoded as vertices. The links that represent a friendship and a forum

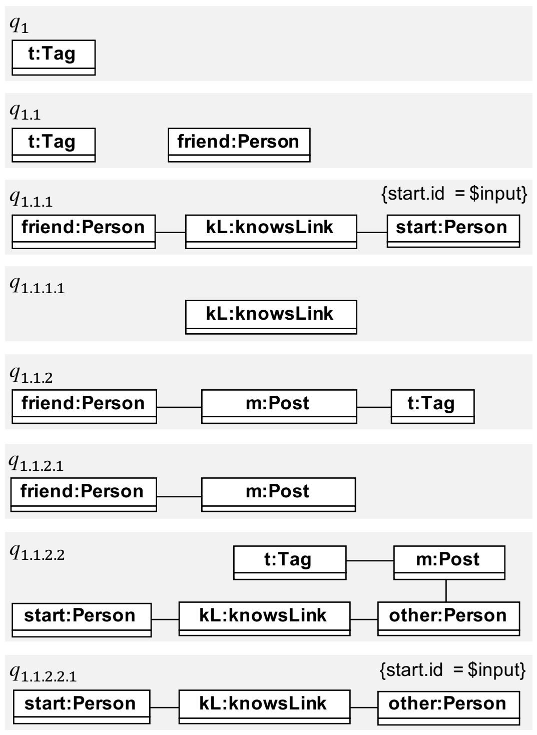


Figure 5.5: Graph patterns used for IC4, where \$*input* denotes an input parameter provided to InTempo

membership are relevant to IC4 and IC5 and have been modeled as vertices with a creation and a deletion timestamp in Figure 5.4—see `KnowsLink` and `hasMemberLink` vertices.

Based on these patterns, we gradually compose the query in $\mathcal{L}_T$ for IC4. Note that the naming scheme of the patterns is based on their nesting level and time is assumed to be tracked in seconds. We search for `Tag`s and `Person`s ($q_{1.1}$) that satisfy the following three conditions simultaneously. First, they are friends with the start `Person` ($q_{1.1.1}$, where the start `Person` is a user input captured by the pattern constraint). For this condition, it is additionally required to locate the *first time point* where the friendship was created. In MTGL, this may be achieved by the construction:

$$\exists(q_{1.1.1}, \neg\blacklozenge_{(0,\infty)} \exists\, q_{1.1.1.1})$$

where we leverage the knowledge that the `KnowsLink` will be the last vertex created in $q_{1.1.1}$, i.e., after the two `Person`s. The second condition requires friends to have posted `Post`s with `Tag`s where the first time point of the `Post` (its moment of creation) was within the last year:

$$\blacklozenge_{[0,1y]} \exists(q_{1.1.2}, \neg\blacklozenge_{(0,\infty)} \exists\, q_{1.1.2.1})$$

Note that we abbreviate the number of seconds in a year by "1*y*". Finally, these `Tag`s have never been attached to `Post`s of the start `Person` by any other friend:

$$\neg\blacklozenge_{[0,\infty)} \exists(q_{1.1.2.2}, \exists\, q_{1.1.2.2.1})$$

The construction locating the first time point a pattern occurs, that is:

$$\exists(q_{1.1.1}, \neg\blacklozenge_{(0,\infty)} \exists\, q_{1.1.1.1})$$

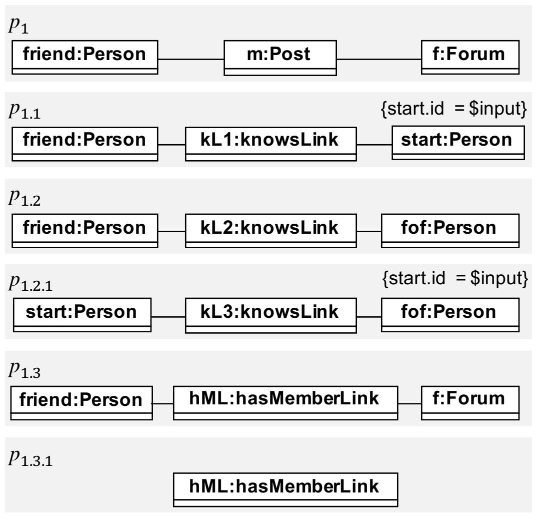


Figure 5.6: Graph patterns used for IC5

uses an unbounded temporal operator which requires that InTempo stores the entire history and effectively disables pruning. As mentioned earlier, the lifespan of a match is always an interval, i.e., a connected set of time points. This characteristic makes the first time point when a match occurs unique, i.e., there can be no two first time points in the past. Therefore, in this particular construction, it is unnecessary to check the sub-condition over the entire history. Instead, the interval of the operator can be reduced to a minimal interval, i.e., $\blacklozenge_{(0,1)}$, which returns the same result while allowing InTempo to avoid storing the entire history. In the following, we abbreviate this construction by the operator *exists-first*:

$$\exists(q_{1.1.1}, \exists^f q_{1.1.1.1})$$

In summary, in $\mathcal{L}_\mathrm{T}$, IC4 is captured by the query IC4 ≔ $(q_1, \psi_{\mathrm{IC4}})$ with ψ_{IC4}:

$$\begin{aligned}&\exists(q_{1.1},\\ &\quad \exists(q_{1.1.1}, \exists^f q_{1.1.1.1}) \wedge \blacklozenge_{[0,1y]} \exists(q_{1.1.2},\\ &\qquad \exists^f q_{1.1.2.1} \wedge \neg \blacklozenge_{[0,\infty)} \exists(q_{1.1.2.2}, \exists q_{1.1.2.2.1})))\end{aligned} \tag{5.1}$$

The patterns for IC5 are shown in Figure 5.6. The (slightly adjusted) query reads: *"Given a start* `Person`*, find the* `Forums` *which that* `Person`*'s friends and friends of friends (excluding start* `Person`*) became members of in the two months before the friendship was created. Return all* `Posts` *in the* `Forums` *created by the start* `Person`*'s friends or friends of friends within that period."*. In $\mathcal{L}_\mathrm{T}$ it is captured by the query IC5 ≔ $(p_1, \psi_{\mathrm{IC5}})$ with ψ_{IC5} :

$$(\exists p_{1.1} \vee \exists(p_{1.2}, \exists p_{1.2.1})) \wedge (\blacklozenge_{[0,2m]} \exists(p_{1.3}, \exists^f p_{1.3.1})) \tag{5.2}$$

Note that the number of seconds in two months has been abbreviated by "$2m$". The formulations of IC4 and IC5 in ITQL are available in Section B.3.2.

5.3.1 *Input Logs*

We have used the data generator of the SNB to generate data for two scale factors: sf-0.1 and sf-1, which create a network of 1.5K and 11K `Persons`, respectively. In total, 328K vertices and 1.5M edges are created in sf-0.1, and 3.2M vertices and 17.3M edges in sf-1. The generated data span a period of ten years, from 2010 to 2020. We have captured the creation and deletion timestamps of insert and delete operations into log files of timestamped events. For our experiments, forum memberships and friendships are also encoded as vertices and, therefore, the relevant inserts and deletes in the generated data are similarly represented by events which create or delete instances of `HasMemberLink` and `KnowsLink`, respectively. This brings the total of vertices and edges created by the log for sf-0.1 to 900K and 3.4M, respectively. The log for sf-1 creates 10.2M vertices and 38.4M edges—see the overview in Table 5.1. The logs are available in [134]. The E2P specification for log events is available in Section B.3.1.

The generated data contains two stages: the *functioning* stage which entails the creation of the entire network and a small percentage of deletions, spanning from 2010 to 2013, and the *shutdown* stage which contains only deletions and destroys the network, spanning from 2013 to 2020. We have added a *start-up* stage to the beginning of the log which creates the static part of the network, e.g., `Tags` and `Countries`, at the beginning of the epoch (timestamps 0 to 7).

5.3.2 *Experiment Design*

We envision a scenario where the answers to queries IC4 and IC5 are utilized to detect trends, and to provide a member of the network with dynamic recommendations or warn the member for suspicious behavior in their network when the member logs in. Recommendations can be built based on the returned `Tags` from IC4 and warnings can detect abnormally many new memberships by new friends, returned by IC5. Therefore, in our experiments we evaluate the queries *periodically* on each (simulation) day, which simulates a daily login on the network by the member.

According to the typical SNB execution scenario [103], we first process a large number of operations of the operational stage (the first 35 months) such that a large starting RTM^{H} has formed before the queries are evaluated. This *initial phase* of the experiment corresponds to roughly 800K events in sf-0.1 and 8.8M events in sf-1. The queries are evaluated once per day for the remaining month in the operational stage. After the operational stage, the shutdown stage comprises numerous bulk deletions which span the remaining period, i.e., 7 years, and destroy the network. This would not constitute a realistic setting for the scenario, as only deletions would be processed. Hence, the experiments only run until the beginning of the shutdown stage. The percentage of the elements that are deleted in the operational stage is shown in Table 5.1.

We evaluate the performance of IT (no pruning) and IT$_{+P}$ (with pruning). IT$_{+P}$ is only executed for IC5, as IC4 contains an unbounded past operator, i.e., it refers to the entire history. Depending on the log, each query is evaluated for a different start person. This was done to ensure that the start person would be actively involved in query evaluations. To choose the start person, we created a

Table 5.2: Query evaluation time (cumulative) for IC4 and IC5 (secs - rounded)

	IC4		IC5	
	sf-0.1	sf-1	sf-0.1	sf-1
IT	9	77	11	169
IT_{+P}	-	-	10	174

list that sorted network members according to their number of friends (larger to smaller) and randomly picked a person from the top half. The log sf-0.1 is executed for the person with `id`= 483 and sf-1 for the person with `id`= 361.

In the initial phase, before the periodic evaluation begins, the TVGDN is populated with all matches in the starting graph. The first execution of the periodic phase updates these matches. Each variant is executed for each input log and is measured for either query evaluation time or memory consumption. Experiments which measure time were executed 10 times and the average values are reported. The experiments are conducted on the same workstation as all other experiments.

5.3.3 *Results*

Table 5.2 shows the cumulative query evaluation times for both IC4 and IC5. The initial phase for IC4 and IC5 over sf-0.1 lasted approx. 18 and 39 seconds, respectively. Over sf-1, the initial phase for IC4 and IC5 lasted 145 and 363 seconds, respectively. The effect of pruning in IT_{+P} is marginal as only a few deletions, i.e., less than 5%, occur in the log—see Table 5.1. On the other hand, the overhead of pruning is also reduced. Given that no pruning occurs for the first 35 months, a relatively lengthy pruning takes place after the first execution in every experiment. For instance, the first pruning for sf-1 lasts approx. 47 secs. The average duration of all other executions of pruning is 287ms. Nevertheless, the accumulation of this overhead combined with the small percentage of deletions, causes IT_{+P} to require more time than IT over sf-1.

Figure 5.7 and Figure 5.8 show the query evaluation time for IC4 over sf-0.1 and sf1, respectively, in detail; Figure 5.9 and Figure 5.10 show the query eval-

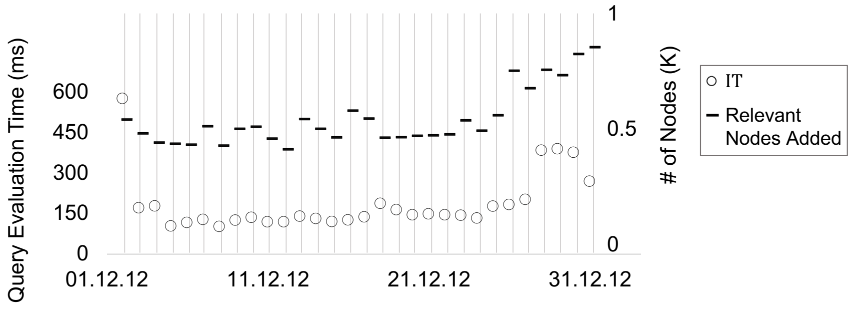


Figure 5.7: Query evaluation time (IC4 - sf-0.1)

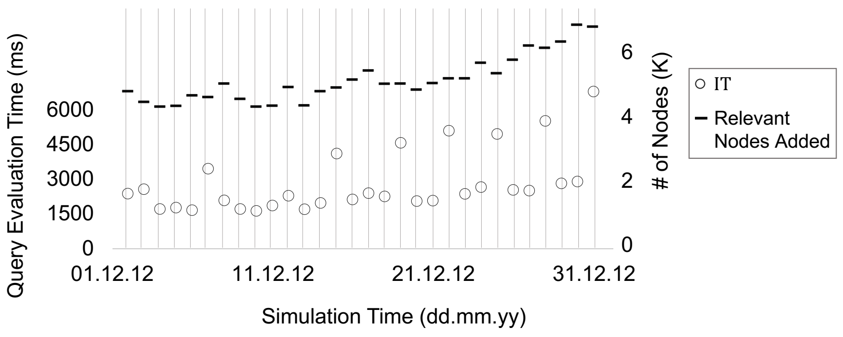


Figure 5.8: Query evaluation time (IC4 - sf-1)

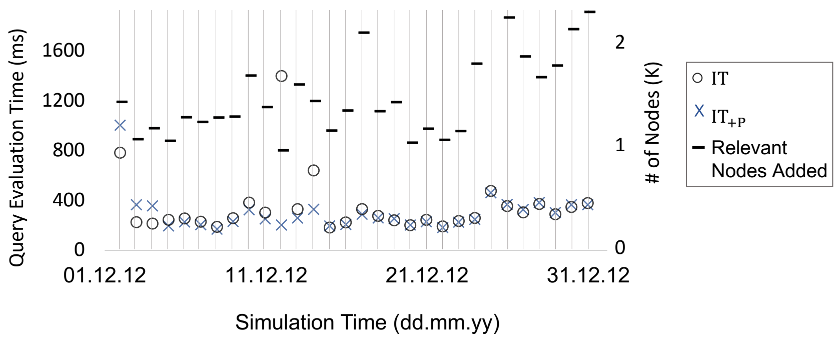


Figure 5.9: Query evaluation time (IC5 - sf-0.1)

uation times for IC5. The figures also show the number of *relevant nodes* (in thousands) added per period: Relevant nodes are those included in the patterns of the TVGDN nodes constructed for the queries, e.g., for IC5, instances of `HasMemberLink`, `Person`, `KnowsLink`, and `Post`. Generally, larger execution times correspond to periods with a larger number of relevant nodes. On average, over the larger sf-1 log and for IC5 that allows for pruning, the InTempo variants handled 15K additions of relevant nodes per period; the average query evaluation time was 5.7 secs with IT and 5.8 secs with IT_{+P}.

Table 5.3 shows the memory consumption of the two variants. Due to the small number of deletions, the memory consumption of IT_{+P} is only slightly decreased. We measured that only loading the RTM^{H} in memory consumed 2.6GBs for sf-0.1 and 44.4GBs for sf-1.

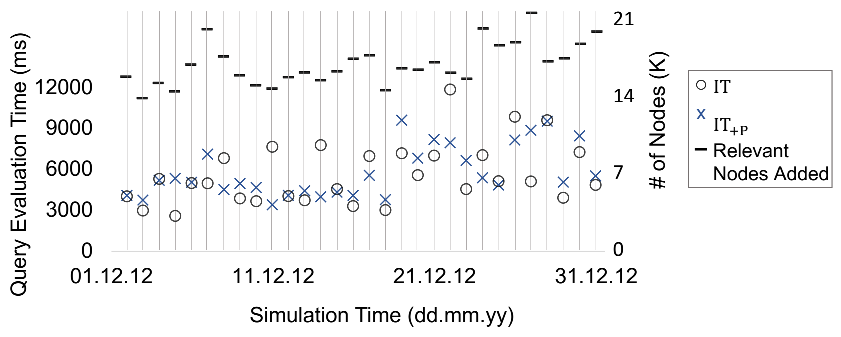


Figure 5.10: Query evaluation time (IC5 - sf-1)

Table 5.3: Memory consumption (max) for IC4 and IC5 (MBs)

	IC4		IC5	
	sf-0.1	sf-1	sf-0.1	sf-1
IT	7040	72389	12631	133880
IT_{+P}	-	-	12158	130384

5.4 COMPARISON TO STATE-OF-THE-ART

InTempo processes a sequence of events, maintains a history encoding, and verifies whether this encoding satisfies a temporal logic formula. This functionality resembles the objective of RV, where an online algorithm verifies whether a sequence of events representing a system execution satisfies or falsifies a temporal property. The algorithm is required to maintain an (internal) representation of the history of the execution, similar to the RTM^H. We therefore use the state-of-the-art RV tool MonPoly to acquire a baseline for the performance of $IT^{\circlearrowleft}$ and $IT^{\circlearrowleft}_{+P}$ in detecting issues during the activity.

RV tools are typically not intended for usage with structural models. In MDE, storing past versions of a (structural) model may be achieved by a *model indexer*, i.e., a solution which monitors file-based repositories such as Git or SVN, stores models or model elements of interest to a database, and maintains an *index*, i.e., an efficient representation, of the model evolution which is amenable to model-element-level querying [7]. The indexer Hawk integrates a time-aware database and provides temporal primitives which can be used to query the history of a model. Using these primitives, we formulate the queries MG1 and MG2 and compare the performance of Hawk to that of InTempo.

5.4.1 *Runtime Verification with MonPoly*

MonPoly [13, 14] is a command-line tool which notably combines an adequately expressive specification language with an efficient incremental monitoring algorithm. It has been the reference in evaluations of other RV tools [46, 82] and among top-performers in an RV competition [12]. Its specification language is based on *Metric First-Order Temporal Logic* (MFOTL) [13] which uses first-order *relations* to capture system entities and their relationships. MonPoly processes a sequence of timestamped events, maintains an internal representation of the system execution, and checks whether it falsifies a given formula. Unlike an RTM, representations in RV tools are created ad-hoc , and they are pruned by default, i.e., they contain only the data that is relevant to the formula and has not been checked yet.

The semantics of MFOTL are *point-based*, i.e., the logic assesses the truth of a formula only at the time points of the events in a sequence and not for the entire time domain as interval-based logics such as MTGL. Therefore, a result in MFOTL is not accompanied by a temporal validity as in InTempo. Furthermore, for certain formulas, point-based semantics may yield counter-intuitive results which disagree with interval-based semantics—on the other hand, it may allow for more efficient monitoring algorithms compared to those based on interval-based semantics [see 15]. Although the difference in interpretations may affect more extensive evaluations, it does not affect the conditions of the queries discussed in Section 4.4.1.

Encoding a graph pattern in MFOTL requires an explicit definition of the expected (temporal) ordering of the events that corresponds to the order of creation of the elements in the simulation. To emulate pattern matching, we would therefore have to specify an MFOTL formula that would consider all possible events of interest as a start for matching the pattern and then search in the past of the execution or in the present for the rest of the events of interest. Leveraging the knowledge of the actual order in which events occur in the simulation, we simplify the formulas for MonPoly by specifying only the correct ordering. This creates an advantage for MonPoly in the comparison with our implementation which we deem is justified as MonPoly is not intended for pattern matching. For ensuring that a match is returned only if the lifespans of entities overlap, we use a construction suggested by the MonPoly authors [13]: for a creation event $c(a)$ and a deletion event $d(a)$, we encode the lifespan of the entity a by $\neg\exists\, d(a)\, \mathsf{S}_{[0,\infty)}\, \exists\, c(a)$.

5.4.2 *Indexing and Querying the History of an RTM with Hawk*

Hawk [61] integrates a time-aware graph database [80] which tracks changes between (timestamped) repository commits and, therefore, equips Hawk with the capability of querying the history of a model. Hawk represents history by *versions* of types and type instances. A new version of a type is created every time a type instance is created or deleted; the initial version of a type has no instances and types are never removed from the indexer. A new version of an instance is created every time one of its features changes; instances are removed from the indexer when they are deleted from the model. Versions are timestamped based on the timestamp of the repository commit that created the version in question.

Hawk formulates queries in the *Epsilon Object Language* (EOL), which is founded on OCL [127, 159]. Hawk extends EOL with support for temporal primitives such as *time, getVersionsFrom(τ), eventually,* which enable retrieving the timestamp of a version, obtaining a specific collection of versions based on their timestamp τ, or making assertions over a collection of versions. EOL supports methods native to EMF which obtain the container of an instance or the contents of one of its features.

5.4.3 *Query Specifications and Conducted Experiments*

In the following, we use the languages of MonPoly and Hawk to specify the queries in the case-studies. We begin with the queries MG1 and MG2 in SHS, i.e., $\mathrm{MG1} := (u_1, \neg\psi_{\mathrm{MG1}})$ with $\psi_{\mathrm{MG1}} := \Diamond_{[0,3600]}\exists\, u_{1.1}$ and $\mathrm{MG2} := (u_1, \neg\psi_{\mathrm{MG2}})$ with $\psi_{\mathrm{MG2}} := \neg\exists\, u_{1.2}\mathsf{U}_{[0,3600]}\exists\, u_{1.1}$—see Section 4.4.1.

We map MG1, in a straightforward manner to its MFOTL equivalent, i.e., the language of MonPoly. The pattern u_1 is enclosed by an existential quantifier, other operators remain intact, and relations are used instead of patterns.

A straightforward mapping is not possible for MG2 however, as MonPoly restricts the use of negation in this case. It does so for reasons of monitorability, as the tool assumes an infinite domain of values, and the negation of the existence of $u_{1.2}$ at a given time point when it does not exist is satisfied by infinite values and is therefore non-monitorable by MonPoly—note that the use of negation is unrestricted for *since*, as with the lifespan construction in Section 5.4.1. In the conducted experiments, we compare to MonPoly only for MG1. The translations in MFOTL are available in Section B.2.4.

We proceed with Hawk. The query MG1 is translated in EOL by obtaining a set c_1 with all `Probe` instances with `status` set to *sepsis*, created within a certain time window. Subsequently, we obtain a collection c_2 with the container of all instances in c_1 (the instance of `PMonitoringService`). For each instance in c_2, we obtain its container (the instance of `SHSService`). We collect all contents of the `connected` feature of the `SHSService`, i.e., all the connected services, that are of type `DrugService` in a collection c_3 and, for each instance in c_3, we check whether the contents of its `probes` feature include a `Probe` instance with `status` set to *anti*, whose timestamp satisfies the temporal constraints. The query MG2 is identical except it also checks whether an instance of `Probe` with `status` *release* exists in the period between an instance with `status` *anti* and an instance with `status` *sepsis*. The EOL queries are available in Section B.2.5.

Attempting to evaluate MonPoly and Hawk with the SNB logs results into considerable practical difficulties. In the SHS case-study, we optimize pattern matching by MonPoly by arranging the ordering of relations in the MFOTL property, leveraging the knowledge on the order in which these events occurred in the simulation. Applying the same optimization for SNB is impossible as there is no fixed order in which some patterns could occur, e.g., the friendship and the membership in $q_{1.1.2.2}$. Therefore, the MonPoly properties would have to feature many alternative orderings which would deteriorate the performance of the tool. For Hawk, the initial phase of the experiment required generating and indexing approx. 800K and 8.8M (large) models for sf-0.1 and sf-1, respectively. Saving these models as XMI files and indexing them was quite slow: only a few tens of thousands of models had been processed after several hours. These difficulties indicated that the tools are not meant for this setting and, using them nonetheless would compromise the conclusions drawn by the experimental evaluation. Therefore, the tools are excluded from the SNB experiments.

5.4.4 *Input Logs*

Regarding MonPoly, we encoded the SHS metamodel by relations, following generally standard practices [see 124]. We translated all simulated logs, i.e., *real*, x10, x100, into sequences of events based on this encoding. An overview of the translated logs is shown in Table 5.1—translations are prefixed with an "*M*-". The logs are available in [134].

Hawk supports models created in EMF which allows us to re-use the SHS metamodel in Figure 2.1 as well as the E2P specification and the relevant component of our implementation to map events in the log files *real*, x10, and x100 to model modifications. The modifications are identical to those created in the experiments with InTempo.

5.4.5 *Experiment Design*

MonPoly processes the events in the file and, for each event, updates its internal representation of the system behavior and its result. As explained earlier, the representation only retains data which are (temporally) relevant to the formula. This algorithm resembles the experiments for InTempo and, therefore, MonPoly is executed only once per experiment. Each experiment entails the execution of the tool with one translated log and the property MG1—as MG2 cannot be monitored by MonPoly. The latest MonPoly version at the time of writing is used (1.1.10) and run on the same machine as the implementation variants. For measuring the memory consumption and execution time, we use the output generated by MonPoly.

For Hawk, the experiments proceed similarly to those conducted for InTempo—see Section 5.2.2. Each experiment entails the evaluation of either MG1 or MG2 and is measured for either time or memory. Following the processing of an event, i.e., model modifications, the model is saved as an XMI file, i.e., the standard EMF file extension, and committed to a programmatically created Git repository. The timestamp of the commit is set to the timestamp of the event. Hawk is invoked periodically at the same time points of the loop invocations of the InTempo variants. In every invocation, Hawk is first requested to update its index and then to evaluate the query. Given the periodic invocation, queries obtain only the necessary versions of `Probe` such that no adaptation issues are missed. That is, the translations use the primitive *getVersionsFrom*$(\tau - 2\mathcal{W}^r)$, which retrieves all instances from τ up to $2\mathcal{W}^r$, with τ the timestamp of the latest change and $\mathcal{W}^r$ the relevance window —see Section 4.3.1. The latest history-aware Hawk version at the time of writing is used (2.2.0) and run on the same machine as InTempo. Query evaluation time and memory consumption are measured similarly to InTempo. The time required for saving and committing the XMI file is omitted.

5.4.6 *Results*

The results for the cumulative *issue detection time* (in seconds) for the three logs are shown in Table 5.4. The issue detection time refers to the amount of time each tool or variant requires to produce the correct result: for $\text{IT}^{\circlearrowleft}$, this is only the sum of the times for Analyze, i.e., the query evaluation time, for

Table 5.4: Issue detection time (cumulative) for MG1 and MG2 (secs - rounded)

	MG1			MG2		
	real	x10	x100	*real*	x10	x100
$\text{IT}^{\circlearrowleft}$	7	596	68952	8	585	68744
$\text{IT}^{\circlearrowleft}_{+P}$	2	10	113	2	14	147
Hawk	730	57685	>250000	726	57534	>250000
MonPoly	2	283	189638	-	-	-

Table 5.5: Memory consumption (max) for MG1 and MG2 (MBs)

	MG1			MG2		
	real	x10	x100	*real*	x10	x100
$\text{IT}^{\circlearrowright}$	48	268	2479	58	365	3442
$\text{IT}^{\circlearrowright}_{+\text{P}}$	28	28	28	30	30	30
Hawk	194	1201	-	194	1204	-
MonPoly	17	39	269	-	-	-

every invocation; for $\text{IT}^{\circlearrowright}_{+\text{P}}$, however, it is the sum of Analyze and Maintain for every invocation. Similarly, the sum of the indexing and the querying time for every invocation is reported for Hawk. The execution of Hawk over x100 was stopped after almost three days, hence no results are reported.

Issue detection with MonPoly is faster than $\text{IT}^{\circlearrowright}$ for *real* and x10. However, MonPoly is slower than $\text{IT}^{\circlearrowright}$ for x100. $\text{IT}^{\circlearrowright}_{+\text{P}}$ outperforms MonPoly over the larger logs x10 and x100. Besides the suitability for querying structure, another reason is that, as shown in Table 5.5, pruning significantly reduces the size of the RTM^{H} and hence its memory consumption. As a result the time spent for pattern matching is decreased. Hawk is the slowest in detecting issues and the most costly in terms of memory. The size of the database of Hawk on disk was deemed irrelevant and is not reported.

Figure 5.11 (in logarithmic scale) shows the *speedup* of $\text{IT}^{\circlearrowright}$ over Hawk, i.e., (hk/it), with hk the issue detection time of Hawk for an invocation and it the issue detection time of $\text{IT}^{\circlearrowright}$ for the same invocation. The speedup value of 1 is marked by a dashed line. $\text{IT}^{\circlearrowright}$ was faster than Hawk in all invocations except the plot points below the dashed line—which amounts to approx. 1.2% of the total number of invocations. Figure 5.12 shows the speedup of $\text{IT}^{\circlearrowright}_{+\text{P}}$ over Hawk, where once more the speedup value of 1 is marked by a dashed line. Hawk was always slower than $\text{IT}^{\circlearrowright}_{+\text{P}}$. In fact, their difference increases as the simulation proceeds, as pruning in $\text{IT}^{\circlearrowright}_{+\text{P}}$ gradually decreases the size of the RTM^{H}. Plot points where the speedup of $\text{IT}^{\circlearrowright}$ is larger than the speedup of $\text{IT}^{\circlearrowright}_{+\text{P}}$ are invocations in which query evaluation of $\text{IT}^{\circlearrowright}$ took a very small amount of time, i.e., less than a millisecond; although the query evaluation time in these invocations was similar for $\text{IT}^{\circlearrowright}_{+\text{P}}$, the time spent in pruning added an overhead which reduced speedup.

MonPoly only outputs the cumulative time required for monitoring the log; the time per event is not reported, hence a comparison of the speedup was not possible.

5.5 DISCUSSION

We discuss the results of the experimental evaluation with respect to the objective of *increased scalability* compared to the state-of-the-art—see Section 1.3. Then, in light of the experimental evaluation results, we reflect on advantages and limitations of InTempo. We conclude the experimental evaluation with a

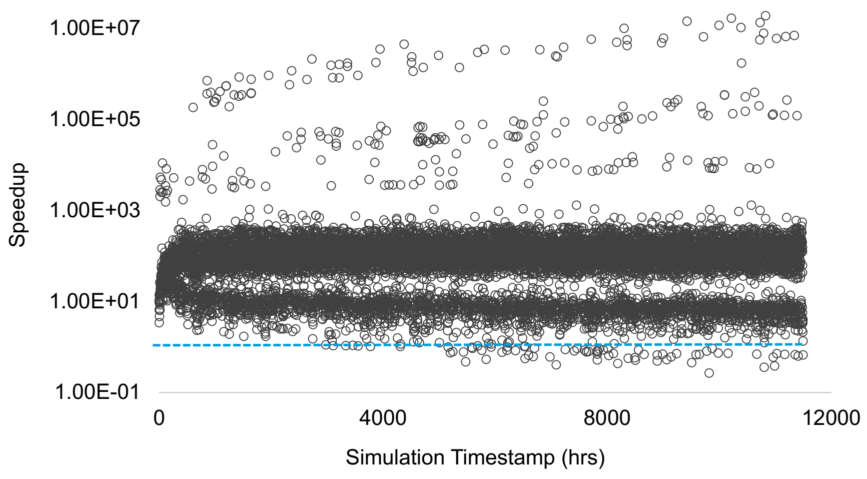


Figure 5.11: Speedup of issue detection time of IT$^{\circlearrowleft}$ over HAWK (MG1 - x10)—values below the dashed line indicate invocations for which IT$^{\circlearrowleft}$ was slower than HAWK

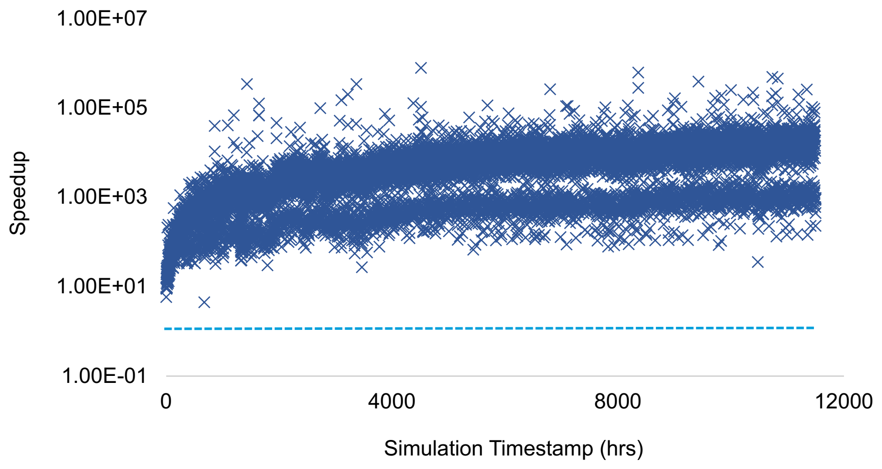


Figure 5.12: Speedup of issue detection time of IT$^{\circlearrowleft}_{+P}$ over HAWK (MG1 - x10)

discussion of the threats to the validity of the experimental evaluation and the measures taken to mitigate them.

5.5.1 *Fulfillment of Objective for Increased Scalability*

We assess the fulfillment of the objective for increased scalability based on the query evaluation times and memory consumption.

Regarding query evaluation times, we perform a comparison among the issue detection times of IT$^{\circlearrowleft}$, that used a complete history encoding, IT$^{\circlearrowleft}_{+P}$, a variant that used a pruned history encoding, and two state-of-the-art tools: the RV tool MONPOLY, with which we emulated pattern matching with relations, and the model indexer HAWK. By issue detection time, we refer to the time required to return sound results, i.e., query evaluation time for IT$^{\circlearrowleft}$, query evaluation time plus pruning time for IT$^{\circlearrowleft}_{+P}$, execution time for MONPOLY, and indexing time plus query evaluation time for HAWK. Generally, both IT$^{\circlearrowleft}$ and

$\text{IT}^{\circlearrowright}_{+\text{P}}$ outperform Hawk, and $\text{IT}^{\circlearrowright}_{+\text{P}}$ also outperforms MonPoly. $\text{IT}^{\circlearrowright}$ is slower than $\text{IT}^{\circlearrowright}_{+\text{P}}$ and MonPoly, however it operates on a complete history encoding, and it is therefore capable of computing the precise validity of an answer over the entire history at any point in time. Hawk, which shares this ability, is slower than $\text{IT}^{\circlearrowright}$. We can therefore conclude that query evaluation times of InTempo scale better than the state-of-the-art. Issue detection times were also *relatively* fast with respect to the invocation frequency in the SHS case-study. Although stronger claims on this aspect require further investigation, these times indicate that InTempo may serve as a basis for more sophisticated adaptation schemes.

We perform a similar comparison regarding memory consumption. $\text{IT}^{\circlearrowright}_{+\text{P}}$ consumes smaller amounts of memory than $\text{IT}^{\circlearrowright}$ while delivering the same results over the entire history. Moreover, owing to pruning, the memory consumption of $\text{IT}^{\circlearrowright}_{+\text{P}}$ remains constant in all experiments. $\text{IT}^{\circlearrowright}$ and $\text{IT}^{\circlearrowright}_{+\text{P}}$ are more memory-efficient than Hawk in all experiments. $\text{IT}^{\circlearrowright}_{+\text{P}}$ is more memory-efficient than MonPoly for larger logs, i.e., x10, x100 which indicates increased scalability.

Except for serving as a baseline for the variant with pruning, $\text{IT}^{\circlearrowright}$ covers other important use-cases where pruning might be infeasible or undesirable. For instance, the queries of interest might change often and thus relevance windows cannot be derived a priori—as historical data might be relevant to another query in the future. Another example is when the incurred cost of pruning on the loop execution time is undesirable. Finally, a third possible scenario is postmortem analysis like self-explanation in self-adaptive systems [19] where the system is required to explain adaptation decisions in its entire history—in our example, $\text{IT}^{\circlearrowright}$ is faster and more memory-efficient than Hawk, which is intended for such use-cases [60, 121]. It should be noted that, given the database back-end and the offline use-cases, minimizing memory consumption may not be a primary focus for Hawk. Moreover, by storing versions of types and type instances, Hawk implicitly stores the history of types, links, and attribute values, which would require a manual encoding in InTempo.

The SNB log files allowed for a more extensive evaluation of InTempo as they involved more complex, realistic, and considerably larger graph structures. $\mathcal{L}_T$ is used for more complex queries than those in the SHS case-study which stem from a domain which is significantly different to healthcare. The execution of InTempo for two scale factors of the benchmark indicates that, for the specific experimental setting, the query evaluation time of InTempo in the smaller scale factor scaled well in the larger scale factor. This evaluation allows for indirect comparisons with other tools in the future—as well as future versions of InTempo.

A GDN stores all intermediate query results. During the network execution, only nodes affected by model modifications are executed which, combined with local search, makes for an incremental evaluation framework which is optimized for fast query evaluations. This feature is emphasized in the SNB experiments where the structure of the RTM^{H}, in contrast to the SHS case-study, is amenable to local search; owing to local search, pattern matching may skip large parts of the RTM^{H} that are unaffected by modifications. Therefore, in relation to the number of events and added vertices (see Table 5.1), the query evaluation times were considerably fast.

On the other hand, given the size of the patterns in SNB queries and the number of matches, storing all intermediate results has an adverse effect on memory consumption. The effect was aggravated by TGDNs featuring many nodes whose LHS is the context pattern.These nodes find and store matches that have already been found and stored by other nodes in the network. For example, IC5 features many such nodes, with some of them redundantly storing matches of elements which are in great quantities in the data, e.g., `Posts`.

5.5.2 *Advantages and Limitations*

From a purely technical viewpoint, adequately expressive RV solutions like MonPoly are capable of representing an architectural state and detecting issues which concern the evolution of the state. Nevertheless, the lack of inherent support for the representation of structure, renders the use of MonPoly in this context problematic: Transferring the architecture to the relation-based representation of MonPoly resulted into an overly technical encoding; matching a pattern comprising events in a relatively efficient manner was only possible because the exact order of the occurrence of these events was known a priori; representing the lifespan of an entity required a technical construction in the formula which deteriorated performance; finally, translating one of the properties which concerned the prohibition of the existence of a pattern into MFOTL was not possible due to the syntax restrictions of MonPoly. These leads indicate that MonPoly and similar RV tools are suboptimal for graph-based representations and graph-based querying, and therefore they cannot replace RTM-based solutions in architectural adaptation schemes.

In the presented experimental setting, InTempo outperformed Hawk and MonPoly. Nonetheless, the level of maturity of the two tools surpass the prototypical implementation we present. Hawk supports various modeling technologies, including EMF. It also seamlessly supports the storage of the history of attributes, links, and types. Although, its database back-end is expected to yield slower access times than an in-memory history representation, it offers other advantages, e.g., increased scalability with respect to the size of the models it could store, which may benefit other settings but were not explored in our evaluation. Both Hawk and, to a lesser extent, MonPoly, support monitoring the evolution of a value, e.g., aggregating the value of an attribute for each query evaluation and monitoring whether the sum exceeds a limit. We plan to address this limitation by equipping the TVGDN with auxiliary nodes used to encode input parameters as constraints that are checked during pattern matching.

As demonstrated by the SNB experiments, InTempo requires that attributes, types, and links whose evolution is of interest are encoded as entities. For the SNB experiments, this meant that two links in the original metamodel were encoded as entities, which roughly tripled the number of vertices in the RTM^{H} compared to the model normally created by sf-0.1 and sf-1. Although this is a customary modeling technique for EMF-based implementations, other tools, such as Hawk, handle the encoding in a manner transparent to the user.

Originally, the creation timestamp in SNB is captured by a vertex or edge attribute to which queries may refer directly. Therefore, the original queries IC4 and IC5 contain timing constraints based on the physical time, e.g., "before

October 2010". We adjusted these constraints to logical time, e.g., "in the last two months", as references to physical time are not currently supported by $\mathcal{L}_T$—and typically are not provided by logic-based languages, e.g., MFOTL. Hawk is able to express such references to physical time. A related limitation is that InTempo opts for the TVGDN construction which is highly incremental but, when monitoring for falsifications of properties that resemble MG1, may return non-definite answers; in order to exclude non-definite answers, InTempo can postpone returning answers whose temporal validity is within a certain period of time to ensure that enough time has passed for the answers to be definite—see Section 4.2.3. This technique is effective when monitoring for falsifications of properties that resemble MG1; however, the technique should not be used when monitoring for satisfactions of such properties, as it imposes a delay in returning answers that are definite. We plan to make Operationalization of InTempo capable of detecting whether an MTGC monitors for falsifications; if not, Operationalization will automatically disable the delaying of answers.

Queries in SNB draw from SQL-based languages and their statements resemble SQL queries. Their translation into a temporal logic like MTGL required a certain level of familiarity with temporal logics and resulted into non-trivial MTGCs. We plan to equip $\mathcal{L}_T$ with constructions that facilitate such translations, e.g., the *exists-first* abbreviation we introduced in Section 5.3. However, there are certain features of SQL-based languages which are typically not offered by logic-based languages, e.g., aggregations and limiting or sorting of results—MFOTL stands out as it offers the capability of numerical operations such as aggregation. Related aspects in the SNB queries have been omitted from our formulation of IC4 and IC5 in $\mathcal{L}_T$.

The SNB experiments indicated that larger TVGDNs may require a significant amount of memory. We plan to investigate whether the context patterns used in nodes can be optimized so that memory consumption is reduced.

5.5.3 *Threats to Validity*

We organize this section based on the types of validity in [163], i.e., *conclusion*, *internal*, *construct*, and *external*.

Threats to Conclusion Validity

Threats to conclusion validity refer to drawing incorrect conclusions about relationships between an experiment and its outcome, for instance, by reporting a non-existent correlation or by missing an existent one.

We mitigated the possible impact of threats to conclusion validity by carefully selecting the experimental data; the log files used were either real (*real* log in the SHS evaluation), synthesized based on real data by employing statistical bootstrapping (x10 and x100), or generated by an independent benchmark (sf-0.1 and sf-1 in the SNB evaluation). Our synthesis method is documented in [133].

Each SHS experiment essentially executed the same query over an increasing event sequence for thousands of times, therefore yielding measurements of an adequately large size. For the SNB, we conducted the experiments measuring the query evaluation time repeatedly and reported the averages of values. Moreover, we studied the benchmark characteristics and attempted to refine statements on the relationship between measurements and conclusions: For instance, in the SNB evaluation, we reported on the number of total additions of relevant vertices instead of the number of events per period.

Threats to Internal Validity

In this context, threats to internal validity are influences which might have affected metrics, i.e., the query evaluation time and the memory consumption. In the following, we describe the measures that were taken to minimize such threats.

The experiments measured these metrics separately and systematically via a controlled simulation of an SHS and the partial execution of a benchmark. Multiple logs were used with an increasing event rate which evaluated the InTempo variants over an increasing load; all other aspects were kept identical: in the SHS case-study, the variants used the same metamodel and the same Monitor, Plan, Execute activities per experiment; similarly, the SNB experiments used the same metamodel, the same event sequences, and the same starting graph per scale factor. Both sets of experiments translated log events into model modifications. The experiments for Hawk used the same logs and translations as InTempo. The experiments for MonPoly entailed the systematic translation of events into relations based on fixed rules.

All experiments were performed on the same machine and, during the experiment, no other (active) task was run on the machine. The values reported in the results of the experiments for InTempo variants and Hawk are based on the values reported by the JVM: The value of free memory was subtracted from the value of total memory. Before measuring memory consumption, we always suggested a garbage collection to the JVM. The values for MonPoly are based on statistics provided by the tool itself, which in turn relies on standard utilities available in UNIX operating systems.

Threats to Construct Validity

Threats to construct validity refers to situations where the used metrics do not actually measure the *construct*, i.e., concept.

In order to minimize threats to construct validity, we used the standard metrics of query evaluation time and memory consumption, measured in conventional ways. We have ensured that these metrics were used in experiments that returned sound results. First, the detection of falsifications in the logs by the variants in the SHS evaluation has been manually double-checked by the author and confirmed by MonPoly and Hawk. Additionally, the answers for the queries in the SNB evaluation have been confirmed by JAVA code. Finally, we have provided formal arguments to substantiate the claims on soundness of computation of temporal validity and the detection of all changes to temporal validity despite pruning the history representation.

Threats to External Validity

Threats to external validity may restrict the generalization of our evaluation results outside the scope of the conducted experiments. In the following, we discuss threats which could influence the generalization of the experimental setting and measurements.

Regarding the experimental setting, we have mitigated threats to external validity by creating the metamodel of the SHS case-study based on the artifact in [161], a peer-reviewed self-adaptation exemplar that has been used for the evaluation of solutions for self-adaptation. Moreover, the simulation used either real data, or data that was synthesized based on the real data, and enacted an instruction from a real medical guideline [126].

Similarly, the SNB experiments were based on the output of an established benchmark which employs sophisticated, well-documented methods for generating realistic data. Queries are similarly designed by experts to expose bottlenecks and stress-test the performance of graph-based technologies. The SNB data was generated using the default parameters, and the metamodel used closely resembles the original—we have only added entities specific to InTempo, i.e., the `MonitorableEntity`, or specific to the evaluation, i.e., links of interest were encoded as entities. We made minor modifications to the data which fixed a few consistency issues, i.e., deletions occurring before creations, which probably stem from the fact that deletions in the output of the benchmark are a rather new feature that is still under development [103, 160].

Regarding the generalization of the performance measurements of InTempo variants, we have conducted two evaluations based on considerably different application domains. The evaluations featured metamodels and models which differed with respect to size and graph characteristics: the SHS data corresponded to a relatively simple star structure which demonstrated the effects of pruning; the SNB data corresponded to a rather large graph with numerous inter-connections which resembled real social networks. The SHS queries showed the advantages of a graph-based temporal logic for architectural runtime monitoring, while the SNB queries explored innovative use-cases of this temporal logic. Our implementation performed solidly in both settings and its results were emphasized via the comparison to MonPoly and Hawk.

We have applied the following optimizations to our implementation. For the removal of elements from EMF models, we transparently replaced the potentially expensive native EMF method with an optimized version. The replacement was done via a JAVA agent which was used for both Hawk and InTempo. The two evaluations were rather different with respect to the evolution of the RTM^{H}: SHS started with a single vertex, whereas SNB started with a significantly large graph. Hence, we configured the Story Pattern Matcher, the pattern matching tool used by InTempo, with a different strategy for search plan generation for the two evaluations. This was done to expedite the experiments and after taking into consideration that the performance of InTempo over the SNB logs would not be compared to Hawk or MonPoly.

On the comparison, we note that MonPoly is not intended for pattern matching. Our emulated pattern matching, although optimized, might have room for improvement. MonPoly relies on a point-based interpretation while InTempo reasons over intervals which can lead to discrepancies between interpretations in more extensive comparisons [see 15]. One of the main reasons MonPoly was chosen for the comparison is the compatibility of MTGL and MFOTL which allowed for a direct mapping of temporal operators and, therefore, reduced the risk of introducing any bias in translations. Based on this mapping, MonPoly could not monitor one of the properties used in the experiments. There might exist equivalent monitorable MFOTL formulas, the syntax of which however would not match that of the MTGC.

We used Hawk in a manner compliant to the examples on the website of the tool [50] at the time of writing. Hawk offers features which could potentially improve the performance of the tool. For example, EOL offers a syntax extension that supports pattern matching. Hawk supports a method to create *annotations*, i.e., predicates over the occurrence of elements or the values of their attributes. Annotations are updated in each indexation, thus rendering the index capable of accessing annotated elements via a lookup—however, annotations have to be defined manually and before the indexation of the model starts. Furthermore, an annotation cannot refer to another annotation, thus, nesting predicates is not supported. Such optimizations, their applicability, and their trade-offs will be investigated in our future work.

6 RELATED WORK

This section discusses work related to our contributions. Based on the problem statement in Section 1.2, we identify the following four basic dimensions in our work: the *MDE* dimension, where an architectural RTM, i.e., a causally-connected structural representation of the state of the architecture, acts as an interface to monitor and manage a system during its execution; the *Time* dimension, where the history is expressly encoded, and temporal queries are specified and evaluated over the encoding; the *Formal* dimension, where concepts are formalized and query answers are shown to be sound; the *Scalability* dimension, where the dependence between the performance of the solution, i.e., query evaluation times and memory consumption, and the size of the history is mitigated. Table 6.1 shows a list of features corresponding to each dimension; the list is based on the objectives and requirements in Chapter 1.

The aim of the thesis is to present a systematic treatment of the problem of evaluating temporal queries over history-aware architectural RTM at runtime. To this end, we have introduced the query language $\mathcal{L}_T$ for the specification of temporal model queries that include quantitative temporal requirements on the evolution of an architectural RTM. Moreover, we have introduced the InTempo querying approach that supports the operationalization and the incremental evaluation of such temporal queries over a compact representation of the history of the RTM, i.e., the RTM^H. Finally, we have introduced a prototypical implementation of InTempo based on Eclipse Modeling Framework.

Our contributions fully support the features in Table 6.1. InTempo is an MDE approach, and therefore inherently supports model queries and architectural RTMs; our implementation is compatible with model-based technologies. Regarding the Time dimension, the RTM^H constitutes a compact encoding of the history of an architectural RTM; $\mathcal{L}_T$ supports past and future temporal requirements, timing constraints,[1] and has interval-based semantics; based on these semantics, we introduce the computation of the temporal validity, i.e., an interval for which a match satisfies the temporal requirements. Regarding the Formal dimension, $\mathcal{L}_T$ is based on the *Metric Temporal Graph Logic* (MTGL) which supports standard logical operators and bindings; we provide proofs on the soundness of query answers and account for the runtime monitoring setting with the introduction of the definite semantics and the effective answer set. With respect to the Scalability dimension, we present a formal operationalization framework which enables the incremental evaluation of a query; moreover, we present a pruning method which discards deleted entities from the RTM^H without affecting the soundness of answers.

In the following, we first discuss work in Section 6.1 which, although not concerned with the evaluation of temporal queries, is related to the foundations of our contributions. We then discuss approaches that support a basic form of

1 As mentioned in Section 2.1.2, we assume a soft real-time system and hence soft timing constraints; hard real-time constraints in the evaluation of graph queries [see 32, 34] is outside the scope of the thesis.

Table 6.1: Dimensions and features for identifying and discussing related work.

Dimension	Feature	Description
MDE	Queries	Specification and evaluation of model queries.
	Structure	Representation of a (structural) architectural RTM.
	Compatibility	Compatibility with model-based technologies.
Time	History	Encoding of history of the execution, i.e., past changes and their timing.
	Temporal	Past and future temporal requirements.
	Metric	Timing constraints in temporal requirements.
	Intervals	Interval-based semantics and (interval-based) timing information in query answers, e.g., the temporal validity.
Formal	Expressiveness	Standard logical operators, e.g., conjunction and negation, and bindings, i.e., the capability of expressing how a given graph that has been previously matched evolves over time.
	Sound	Proofs on the soundness of query answers.
	RM	The consideration of the runtime monitoring setting, i.e., that the history encoding may represent an (incrementally updated) unfinished history, and the provision of suitable, i.e., definite, answers.
Scalability	Incremental	An incremental evaluation technique which avoids the re-computation of query answers after each change.
	Memory	Techniques to avoid performance degradation due to the accumulation of history.

historic graph queries via reactive and incremental evaluation in Section 6.2. Section 6.3 discusses approaches that expressly consider querying the history of an evolving structure. Section 6.4 discusses work from the RV research community. Finally, Section 6.5 discusses the approach known as Complex Event Processing which, although more distantly related to InTempo than RV, when combined with graph pattern matching is capable of checking whether the evolution of a structure fulfills basic temporal requirements. Depending on the extent to which they support the features in Table 6.1, we consider works as closely or more distantly related to our contributions. Closely related work and their support for the features in Table 6.1 are discussed in further detail. An overview of the discussions of closely related work is shown in Table 6.2; *partial support* denotes that a feature is supported by the work to a certain extent, which we clarify further when we discuss the work in question.

6.1 FOUNDATIONS

In the RTM^H, time is captured by entity, i.e., vertex, attributes which are assumed to be present for each vertex in the metamodel, i.e., the type graph. Query evaluations are performed via *Graph Transformation* (GT) rules [53] which write or perform computations with these (distinguished) attributes. This rule behavior is based on foundational approaches where GT rules are extended with a notion of time, e.g., [74] or our own previous joint work in [143].

The behavior of rules in a TGDN varies according to the number of matches found by dependencies. In order to support this varying behavior, which is not covered by conventional GT rules, we employed amalgamated GT rules [26, 69]. This behavior could be supported by rules stemming from other formal frameworks for GT, such as *variability-based rules* [151], which however would require more involved technical notation and explanations for the task at hand.

The language $\mathcal{L}_T$ relies on MTGL for the specification of temporal requirements on the evolution of a structural pattern. MTGL was introduced [68] and extended [143, 144] in our previous joint work. These works formally underpin $\mathcal{L}_T$ as described in Section 2.2.1 and Section 2.3; they present implementations which focus on demonstrating the feasibility of their formal results and do not consider performance aspects, e.g., incrementality. In the joint work in [144], we presented an analysis procedure with preliminary support for runtime monitoring of MTGL: The procedure is based on a translation of an MTGC into an NGC with appropriate attribute constraints; the translation can be adjusted so that NGC satisfaction checks over an iteratively updated graph with history return true either as soon as a falsification is detected or only when it has become definite. When a falsification is detected, the procedure returns the time point of the last update. The result abstracts the interval-based semantics of MTGL into a point-based interpretation which lacks precision. The definite satisfaction and falsification relations introduced in Section 3.3 support runtime monitoring of MTGL directly, i.e., at the level of semantics, and enable the computations of the definite falsification and satisfaction spans. Moreover, besides the three-valued interpretation of the procedure, the relations enable an additional three-valued interpretation based on definite satisfactions and definite falsifications, according to the standard convention [16].

Story Diagrams (SDs) [57, 167] are a visual language that combines UML Activity Diagrams and GT rules to specify pattern matching tasks in an object diagram as well as the creation and deletion of objects and links [66]. The prototypical implementation of InTempo presented in Section 5.1 performs pattern matching tasks based on tool support for SDs [66, 114], developed by the research group of which the author of this thesis is a member.

An approach presented in previous work of the advisor of the author, builds on SDs to define *Timed Scenario Story Diagrams* (TSSDs) [95]. TSSDs support an if-then-else decomposition of complex structural properties which integrates timing constraints. Formally, TSSDs constitute formulas whose semantics, presented in [148], are defined over execution traces consisting of model instances. The semantics do not support queries and, moreover, traces are required to represent finished executions; consequently, TSSDs can perform a definite satisfaction check but support solely postmortem, i.e., offline, monitoring. TSSDs do not compute a temporal validity, although they do evaluate timing constraints by reasoning over intervals. A preliminary monitoring framework with partial support for TSSDs was presented in [65, 147] but the implementation is no longer available. The framework lacks any methods for monitoring a TSSD over an incrementally updated trace or for reducing the memory consumption, which emphasizes its focus on offline use-cases.

6.2 REACTIVE AND INCREMENTAL EVALUATION OF GRAPH QUERIES

In reactive model transformations, a graph representing a structural model is constantly updated by a stream of graph elements or events that are mapped to graph elements; a query is evaluated over the graph after each change and its answers are updated. This setting resembles *streaming* [139] and *active model transformations* [17]. Query results track matches over multiple instances of the graph, and could therefore support basic reasoning over the evolution of the graph. We discuss here solutions which are capable of reactive query evaluation and support other important features of InTempo, i.e., architectural RTMs, incremental evaluation, and the runtime monitoring setting.

The Viatra framework [154, 155] stores a graph which typically represents an architectural RTM in-memory, and features a query evaluation engine which evaluates a query incrementally by decomposing it into sub-queries. Contrary to Viatra, InTempo seamlessly integrates a notion of time in the graph representation, the query language, and the evaluation framework. Viatra uses a decomposition strategy similar to the one for obtaining a GDN called RETE [58] (see Section 2.2.4), whose feasibility for decomposing queries in $\mathcal{L}_T$ and subsequent performance effect on InTempo we plan to investigate. A Viatra extension [152] distributes the pattern matching effort for sub-queries over multiple processing units, which is an interesting future direction for InTempo.

The Viatra-based solution by Búr et al. [33] (see Table 6.2) captures safety properties via graph queries which search for occurrences of prohibited structures, i.e., a match constitutes a falsification of the property. Based on the reactive setting, queries are evaluated after every change to the graph. Therefore, although indirectly, after every change the approach is capable of checking structural requirements on the history of the observed execution. An encoding

Table 6.2: Support for the features in Table 6.1 in closely related work

Section	Authors	Feature											
		MDE			Time				Formal			Scalability	
		Queries	Structure	Compatib.	History	Temporal	Metric	Intervals	Express.	Sound	RM	Incremental	Memory
6.2	Búr et al. [33]	●	●	●	○	◐	○	○	●	●	●	○	○
6.3	García-Domínguez et al. [61]	●	●	●	●	●	●	○	●	○	○	○	◐
	Mazak et al. [111]	●	●	●	●	◐	◐	○	●	○	○	○	○
6.4	Basin et al. [13]	●	◐	○	●	●	●	○	●	●	●	●	●
	Havelund et al. [84]	●	◐	○	●	◐	●	○	●	●	●	●	●
6.5	Dávid et al. [40]	●	●	●	○	◐	●	○	◐	●	○	●	○
	Our work	●	●	●	●	●	●	●	●	●	●	●	●

○: not supported, ◐: partially supported, ●: supported of the history of the RTM and the integration of temporal statements into properties are not supported. Moreover, the solution does not support incremental query evaluation but is capable of distributing the pattern matching effort.

The approach presented by Barquero et al. in [10] analyzes a host graph stored in an in-memory database and extracts a sub-graph that features only elements that are relevant to a given query. Each change to the host graph triggers a new (incremental) analysis and an evaluation of the query over the sub-graph. Thus, the analysis affords incrementality and a form of pruning. However, the analysis does not integrate a notion of time—if it were to be extended to support temporal primitives and an encoding of the graph history, it could be used in conjunction with the timing-constraint-based pruning of InTempo to yield a sub-graph that would contain only elements that are both temporally and structurally relevant. The authors provide no formal proofs on the soundness of the analysis.

6.3 QUERYING THE HISTORY OF AN EVOLVING STRUCTURE

In the following, we discuss solutions with inherent support for queries on the history of an evolving structure. Querying the history of elements in a structure is typically achieved by temporal extensions of the *Object Constraint Language* (OCL) [127, 159]. The OCL allows for the specification of structural constraints on model elements. Temporal extensions enable the specification of temporal constraints by integrating standard primitives from temporal logic, e.g., *always* and *eventually* [92, 128, 145, 165], and timing constraints [45] in OCL.

The solutions which focus on storing and querying the history of an RTM, i.e., Hawk [61], to which InTempo was compared, and the solution by Mazak et al. [111] (extended from [71]), similarly introduce primitives for their languages (both based on OCL) which facilitate the expression of temporal constraints over the entire history on both a model and a model element level. Contrary to InTempo and the solution by Mazak et al. [111], Hawk requires no metamodel modifications for storing history. Hawk and the solution by Mazak et al. [111] store history on a database: Hawk uses the time-aware graph database introduced in [80] and the solution by Mazak et al. [111] uses a time-series-database. Querying the history at runtime entails the appropriate derivation of database queries based on the introduced primitives. None of the solutions provides a formalization of the derivation or the evaluation of temporal constraints over the history of the RTM.

Hawk uses the *Epsilon Object Language* (EOL), which is based on OCL. We have opted for a comparison to Hawk, because of the expressiveness the primitives introduced into EOL in [61] afford, i.e., they allow for temporal assertions, e.g., *eventually*, and version traversal, e.g., *earliest*, which retrieves the earliest version in a given collection, and *time*, which retrieves the time point of a given version. These primitives can also encode timing constraints. On the other hand, the OCL-variant by Mazak et al. introduces primitives which focus on the history of values, i.e., that can retrieve the value of an attribute at a certain version or between versions, rather than the evolution of structures. In [61] It is (informally) demonstrated, that the temporal primitives of EOL can express the well-known specification patterns by Dwyer et al. [49], which García-Domínguez et al. demonstrate is not possible with the primitives introduced by Mazak et al. Analogously, without overly intricate encodings, it would not be possible to translate the EOL queries for the SHS case-study (see Listing B.6 and Listing B.7) into the OCL-variant introduced by Mazak et al. We have therefore marked its support for the features Temporal and Metric in Table 6.2 as partial.

Another argument for the choice of Hawk is that the solution supports optimizations for the query evaluation. As discussed in Section 5.4, Hawk maintains an index for the model evolution which may improve querying performance; moreover, Hawk supports the deletion of elements, i.e., the removal of an element from the index and the database. The history of deleted elements is not tracked, which corresponds to a form of pruning. We made use of this feature in our experimental evaluation, whose results however seem to indicate that deletion does not work as expected—hence the Memory feature is marked as partially supported in Table 6.2.

Regarding the database back-end of the two solutions, as the evaluation in Section 5.4 indicated, database accesses take a significant toll on real-time querying performance, especially for far-reaching past queries which are likely to factor in multiple model or element instances in their runtime computations. Notably, Hawk offers the capability to annotate such queries manually prior to the execution, such that their matches are pre-computed while the system is being executed, which, however, is automatically accomplished by the TGDN in InTempo. Moreover, annotations cannot refer to other annotations, thus nesting annotated predicates is not supported. Compared to an in-memory representation, databases have other advantages, e.g., the increased scalability with respect to the size of the models that can be stored. Notably, the solution by Mazak et al. offers the option of clustering processing nodes at the database back-end which increases the scaling potential. Hawk and the solution by Mazak et al. conveniently support attribute-level history (contrary to InTempo which requires for attributes to be encoded as entities in the metamodel, see Section 5.5.2) and on-the-fly operations on attribute values, e.g., aggregation. However, these features also increase the number of database accesses during pattern matching.

Similarly to the TSSDs discussed in Section 6.1, the language presented in [129] builds on SDs to specify a visual, informal notation for the specification of temporal queries with timing constraints. This notation is not accompanied by an operationalization framework or tool support.

The history of an RTM may be encoded via model versioning based on a (general-purpose) *model repository*, e.g., the solution by Haeusler et al. [76]. There is however a considerable difference between the objectives of InTempo and those of model repositories. For instance, branching is seamlessly supported by repositories, whereas, although it can be supported by an RTM^H [see 6], it is beyond the scope of the thesis. On the other hand, for repositories it is often assumed that queries will mostly concern a single timestamp, i.e., a specific version; repositories are therefore optimized for such queries and are less suitable for the on-the-fly evaluation of pattern-based queries that refer to a period of time—as Haeusler et al. acknowledge.

InTempo can be used in conjunction with model-based architectures and systems which rely on evaluating temporal queries. The model-driven architecture for real-time systems presented in [89] is capable of modifying a user-defined model such that code generated from the model emits events when situations of interest occur in the execution, e.g., a state change. These events are gathered into traces, whose compliance to timing constraints specified in the system model are monitored at runtime by external monitoring tools. Monitoring results are used to steer the system. InTempo can be used in conjunction with this architecture to perform the monitoring, analogously to the role of the approach in a self-adaptive loop in Section 4.4. Given its support for models, InTempo would allow for defining temporal requirements and interpreting the results directly on the model level. A system that captures the changes of an RTM into a graph that can be queried postmortem to support explanation of behavior is presented in [125]. As the authors suggest, if integrated with this system, InTempo could enable the provision of explanations at runtime.

Implementations of *time-aware graph databases* typically build on a database back-end and introduce extensions which integrate a notion of time, e.g., [41, 110] and the one in [80] used by Hawk. Back-ends with native support for graphs are capable of generating a push notification when a node or edge that meets certain conditions has been added or modified [3, 140]. Although this functionality provides a certain degree of support for reactive settings, none of the implementations supports incremental evaluation of queries with complex patterns.

Solutions for *temporal graph processing*, e.g., [36, 78, 94, 98, 115] may also be seen as related to InTempo, since they are also concerned with the efficient storage of a graph that is constantly updated and the reactive evaluation of historic queries [see 21]. These solutions represent history as a sequence of change-based snapshots stored either entirely or partly on disk. This allows for the storage of very large graphs but imposes an overhead on queries and limits their applicability in online scenarios. Moreover, models are not the primary focus of these solutions which arguably render their utilization in an MDE context cumbersome.

6.4 RUNTIME VERIFICATION

Seen in a broader context, InTempo processes a sequence of events and verifies whether this sequence satisfies a temporal logic formula. This approach to runtime monitoring resembles the focus of the research community known as *Runtime Verification* (RV). In RV, an online monitoring procedure incrementally processes a sequence of timestamped events and checks whether the prefix of the observed sequence satisfies a temporal property. The procedure is required to maintain an (internal) representation of the relevant history of the execution. Properties may be specified using various formalisms, e.g., temporal logics and regular expressions [11]. In general, the various RV approaches are difficult to compare [see 87, 123], as the different application domains of RV impose specific requirements regarding expressiveness, efficiency, and usability. In the following, we focus on logic-based approaches.

As indicated by the experimental evaluation, replacing RTM-based solutions in architectural adaptation schemes by typical RV solutions is impractical. An RTM is a causally-connected snapshot of the architecture, which therefore may contain entities which are not relevant to properties; the model has to be accessible to end-users or other model-based technologies, as it acts as an interface to monitor and manage the system. Conversely, representations of the system state in RV are created ad hoc and are typically inaccessible during monitoring. Moreover, models are exploited via queries which implies query semantics, i.e., at any point in time, the answer set contains all matches in the RTM that satisfy the temporal requirements. Monitoring algorithms in RV perform a satisfaction check for a temporal logic formula which typically returns an answer from the Boolean domain; moreover, the algorithms are incremental and discard parts of the state representation which have already been checked. Despite these fundamental differences, we discuss these solutions in detail, and opted for a comparison with one, due to the technical similarity between evaluating temporal requirements on the evolution of a pattern in an evolving RTM and the objective of RV.

We compared the performance of InTempo to MonPoly by Basin et al. [13, 14], a well-established RV tool which was among top performers in an RV competition [12]. MonPoly is a command-line tool which notably combines an adequately expressive specification language for the use-cases discussed in the thesis with an efficient incremental monitoring algorithm. Its language is based on the *Metric First-Order Temporal Logic* (MFOTL) [13] which uses first-order *relations* to capture system entities and their relationships. MFOTL supports past and future temporal operators with timing constraints. Moreover, the logic supports bindings and a relation-based encoding of graphs. Nevertheless, as also discussed in Section 5.5, transferring complex structures to such an encoding rendered the latter overly technical; emulating pattern matching was cumbersome even after optimizations; expressing the AC of the query MG2 in the experimental evaluation, which prohibited the existence of a pattern, was impossible as MonPoly restricted the use of negation in this case—see Section 5.4.3. Hence, we mark the support for structure by MFOTL and MonPoly in Table 6.2 as partial. The semantics of MFOTL are point-based which means the logic cannot support the computation of a temporal validity or, as shown in Section 5.4.1, represent the lifespan of a match straightforwardly. Finally, point-based semantics and the interval-based semantics of MTGL can lead to different interpretations for the same properties [see 15] this did not occur in the experimental evaluation of the thesis but may affect other use-cases.

The tool DejaVu by Havelund et al. [84, 85] can monitor properties specified in a first-order metric past-only logic with point-based semantics. Translating MTGCs in the DejaVu specification language would require emulating graph-based encodings (similar to MonPoly) and, moreover, reformulating MTGCs such that they feature only past operators. Such reformulations are not always possible in the presence of graph bindings with the standard temporal logic operators; even when possible, reformulations could be significantly less compact [83, 102]. Additionally, Havelund et al. report that, currently, only timing constraints which span approx. 60 time units or less yield acceptable performance [84]. This restriction renders DejaVu unsuitable for the application scenarios targeted by InTempo.

Recently, the algorithm in MonPoly has been formally verified via a thorem prover [141] and extended with recursive rules [166] which increases expressiveness. An extension with rules has also been presented for DejaVu [86]. These extensions are a promising future direction for InTempo but beyond the scope of the present discussion.

Other logic-based approaches typically provide no or only partial support for key features of InTempo, e.g., events containing data, inherent or indirect support for graphs and bindings, and temporal operators with timing constraints. Monitoring algorithms for interval-based logics with metric timing constraints involve interval computations which, although inapplicable to a graph-based first-order setting, are similar to ours, e.g., [15] which concerns a propositional past-only logic and [107] which entails a three-valued interpretation for a propositional logic over signals. Havelund et al. present a runtime monitoring approach for a logic defined over intervals [81], based on the interval algebra in [1]. Properties in the logic refer to interval relations, e.g., requiring that two intervals overlap, where the intervals my contain data. The logic supports quantification over intervals but does not support quantification over the data.

Few solutions stemming from RV are based on an MDE context. These solutions do not support key features of InTempo: The Viatra-based solution in [33], discussed also in Section 6.2, does not support history or time; the solution by Dou et al. [46] (supported by the tool in [47] and extended in [48]) presents a pattern-based RV technique which concerns propositional events, i.e., containing no data, and is thus unable unsuitable for the use-cases of interest; the solution in [29], which is based on the work by Dou et al., concerns a considerably different setting where properties refer to the behavior of signals, e.g., oscillation, and is focused solely on offline use-cases.

Approaches to (logic-based) architectural RV, i.e., the analysis of the behavior of a system using logic-based RV on the architecture level, typically instrument the system architecture such that it can emit events and then monitor properties over the sequence of emitted events. The work in [150] presents an offline approach where propositional events are stored in a database; properties are specified over these events in an interval logic which is then manually translated into database queries; the query result consists of events which falsify the property. The work in [42] considers a logic similar to MFOTL, hence the presented approach has the same shortcomings discussed for MonPoly, e.g., lack of inherent support for representation of structure and interval reasoning. The work in [109] presents an approach based on EMF, which however does not support timing constraints. Timing constraints are not supported by the work in [70] either, which however focuses on RV of a temporal logic for SASs where a special operator, introduced in [164], facilitates the specification of properties for self-adaptive software; this is also an interesting future direction for InTempo and MTGL.

6.5 COMBINING STRUCTURAL QUERIES WITH COMPLEX EVENT PROCESSING

The approach known as *Complex Event Processing* (CEP) focuses on processing a stream of events and detects patterns based on the content, the ordering, and the timing of events. Detected patterns generate *complex events* which form another stream and can be further processed [see 39]. Outwardly, the objective of RV is similar to that of CEP, the two however have fundamental differences. Of interest to the present discussion are the following: Languages used in CEP are not based on temporal logic (in fact, they often lack formally precise semantics [73]) which makes a direct comparison difficult; the capability of CEP to evaluate sequential patterns is typically limited, while of central importance to RV [77]—we refer to [77] for a more detailed discussion. Therefore, RV is considerably more relevant to InTempo.

We discuss here solutions which combine structural queries with CEP, thereby enabling the evaluation of basic temporal requirements on the evolution of an incrementally updated structure. The solutions by Dávid et al. [40] and Ehmes et al. [52] incrementally evaluate structural queries after each change to a model and generate events when matches are found; they then use CEP to check whether the matches found adhere to a given (event) pattern. While Ehmes et al. rely on a commercial CEP engine, Dávid et al. introduce a novel engine whose formalization is presented in [40]—thus considered more relevant to our work and displayed in Table 6.2. The engine relies on VIATRA for pattern matching—see Section 6.2. The solution has the advantage that models need not be modified to enable checking whether temporal requirements are fulfilled; however, compared to a logic-based approach with support for bindings, the solution is limited in its ability to express temporal requirements on sequential patterns.

The work in [121] presents a solution which combines CEP with Hawk, discussed in Section 6.3. The CEP engine processes events in a log and filters them according to patterns of interest or a sample rate, thereby reducing the size of the history indexed by Hawk. Queries over Hawk are then used to derive explanations on the reasoning of a system that uses Reinforcement Learning.

The solution by [8], later extended in [9] which was discussed in Section 6.3, stores a stream of data as a graph (on disk), and executes graph queries to detect event patterns. The solution introduces the notion of a spatial window to restrict the size of the graph and thus the search space for pattern matching. However, queries executed in the restricted search space may yield inaccurate results and is therefore inapplicable for the use-cases of interest.

The solution by Song et al. [146] presents an alternative approach where the graph is assumed to be already present and events are timestamped edges; the task of CEP is then mapped to the task of processing the edges and matching more complex graph patterns. The presented graph pattern matching technique is enhanced with a timing window which takes the timestamps of edges into consideration and supports partial orderings. However, the technique is approximate and does not guarantee the soundness of query answers.

7 CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

In this thesis, we aimed to address the lack of a systematic treatment of evaluating temporal queries over history-aware architectural runtime models, such that query answers can be used for architectural monitoring and adaptation at runtime. Specifically, our objectives were to contribute an appropriate formalization of the problem, thereby increasing the confidence on the soundness of query answers, and a querying approach with increased scalability with respect to the state-of-the-art. We considered the following requirements, deemed by the relevant literature to be key: a compact history encoding which captures the lifespans of model entities, an adequately expressive query language, the provision of sound answers, incremental query evaluation, and measures for reducing the memory consumption in the face of accumulating history. Based on these requirements, we presented the following contributions.

Regarding the requirement concerning the history encoding, conforming to the common practice of representing an (architectural) runtime model as a typed attributed graph, we introduced the runtime model with history (Section 3.1), where entities have timestamps marking their creation and deletion time points; entities in this model are not deleted by default. Thus, the model encodes previous versions in a single model instance, i.e., in a compact manner.

Regarding the requirement for an adequately expressive query language, we introduced a language for model queries (Section 3.2) which incorporates an interval-based temporal logic defined over graphs to enable the formulation of temporal queries. These queries allow for expressing requirements on the ordering and timing in which changes in the runtime model with history occur. Model queries in this language characterize graph transformation rules, which allows us to employ the well-established theory of (typed attributed) graph transformation. Query answers pair matches with their temporal validity, i.e., the interval for which a match exists and satisfies the temporal requirements.

Regarding the requirement for the provision of sound answers, we provide proofs on the soundness of an answer to a temporal query as well as the method of the answer computation. Notably, for queries with future timing constraints, we present an answer set that accounts for the case where queries are evaluated over an evolving model, i.e., an unfinished system execution. In an unfinished execution, the evaluation over a given snapshot is followed by evaluations over snapshots spawned by future changes; as in our context matches may trigger adaptations, the query answer has to remain unaffected by future changes. This answer set is based on a novel three-valued interpretation of the temporal logic (Section 3.3) used by the language.

Regarding the requirement for incremental evaluation, we present a reactive querying approach, named InTempo, consisting of two core operations: Operationalization and Evaluation. Operationalization (Section 4.1) is based on a novel framework for decomposing a query in the introduced language to a network comprising sub-queries. This network is executed by Evaluation (Section 4.2) in the right order and is amenable to incremental execution, i.e., a sub-query only updates its answers if a change in the runtime model with history affects it. Evaluation uses a technique that allows for a higher degree of incrementality, which however may delay the answers for certain queries.

Regarding the requirement for memory-efficiency, we extended InTempo with an optional operation, called Maintenance (Section 4.3), whereby entities in the model that are not relevant to query evaluations are pruned. We show that pruning does not affect the soundness of answers over a sequence of instances of the runtime model with history. Maintenance aims at reducing the memory footprint in executions where storing the entire history is unnecessary.

We presented an integration of the operations of InTempo into the well-known MAPE-K loop (Section 4.4), thereby obtaining a reference adaptation engine capable of performing history-aware adaptation.

We presented a prototypical implementation of InTempo in Section 5.1.

Our contributions combined graph transformation, a widespread formal underpinning for structural models, with a graph-based temporal logic to enable the specification and evaluation of temporal queries. We equipped each entity, and therefore each match of a query, with a lifespan, i.e., an interval for which the match exists. Exploiting the interval-based semantics of the logic, the evaluation entails interval computations which yield an interval for which a match in the model satisfies a given temporal formula. We hold that the combination of formalisms for structure and temporal behavior, extended by the introduced interval-based reasoning, provides an appropriate formalization of the problem of evaluating temporal queries over history-aware architectural runtime models.

We developed two variants of the implementation: one where Maintenance was enabled, i.e., the pruning variant, and one where it was disabled, i.e., the non-pruning variant. The non-pruning variant operates on a complete history encoding and thus its answers concern the entire observed history, although at the expense of query evaluation times. We evaluated the two variants based on two case-studies: a simulation of an adaptation scenario for an envisioned smart hospital system according to a real medical guideline and both real and synthetic data, and a simulation of the operation of a social network based on data generated by an independent benchmark. We compared the query evaluation times and memory consumption of the variants to two state-of-the-art tools from model-driven engineering and runtime verification with the capability of evaluating temporal queries over the history of an architectural model. The pruning variant yielded the fastest query evaluation times; the non-pruning variant was faster than the state-of-the-art solution from model-driven engineering which can similarly return answers over the entire history. Regarding the memory consumption, the pruning variant performed better than the state-of-the-art in larger logs; again, the non-pruning variant performed better than the model-based solution. We thus hold that the objective for increased scalability was also fulfilled in the presented experimental settings.

In conclusion, architectural runtime models constitute a powerful means for monitoring and adapting large systems with dynamic architecture during their lifetime. Representing and querying the evolution of the model, i.e., its history, enable more informed decision-making as well as the application of runtime models in domains where the consideration of temporal data is mandatory. However, prior solutions for the evaluation of temporal queries over history-aware architectural models lack the formal precision required for monitoring and the technical characteristics required for scalable performance, e.g., incremental evaluation. On the other hand, solutions from the research community of runtime verification offer formal precision but present other disadvantages, the most important of which is arguably the inability to represent complex structure inherently. We presented a novel formalization which integrates a well-established formal foundation for representing structure with a temporal logic, thereby enabling formal precision in the evaluation of temporal queries. The formalization severs as the basis of a querying approach which was conceived with scalability in mind; it supports incremental query evaluation and an optional deletion of historical data which are irrelevant to query evaluations. We presented a prototypical implementation of the approach which was compared to the state-of-the-art from model-driven engineering and runtime verification and yielded promising results in the presented experimental settings. These results indicate that our approach paves the way for sophisticated history-aware self-adaptation solutions and constitutes a highly effective technique for runtime monitoring on an architectural level.

7.2 FUTURE WORK

In this section, we discuss short- and long-term future work. We first discuss work related to performance and technical optimizations which may increase scalability further, thereby enabling the application of our contributions to various settings, e.g., systems with limited resources. Then, we discuss how our contributions can be used as the basis for sophisticated history-aware adaptation schemes. Finally, we discuss extensions which may enable history-aware architectural runtime models and the evaluation of temporal queries in larger settings, i.e., systems of systems.

7.2.1 *Technical Optimizations and Performance*

As discussed before, our decomposition strategy is based on generalized discrimination networks, which is a more general variant of the RETE algorithm. The algorithm is already used for graph pattern matching in state-of-the-art model-driven engineering technologies. The two strategies have different characteristics which may have an impact on their performance. In the future, we plan to investigate the applicability of RETE networks for temporal queries and profile their performance.

The experimental evaluation provided a preliminary indication on the number of events per second that the querying approach can handle. We plan to stress-test InTempo in other experimental evaluation settings, which will indicate whether the approach can be used for application scenarios with considerably larger event streams.

One of our short-term plans is to employ indexing structures that can index intermediate matches based on their intervals. Such an indexing structure is expected to improve the performance of sub-queries and thus of the network. A related plan entails the investigation on whether the storage of context patterns in the network, which is now redundant, can be optimized.

Finally, we plan to experiment with more precise pruning strategies for application scenarios with limited resources. As presented here, pruning is based on a time window which includes those deleted entities in a history-aware model that are temporally relevant to query evaluations and is derived based on the timing constraints of the temporal operators in a given formula. This pruning strategy can be extended with an analysis that would examine other criteria, e.g., whether both operands in a conjunction can still be satisfied, that would allow for optimized pruning which in certain cases can be performed without considering the time window. If the overhead of this analysis is increased, another method for the application of pruning, e.g., periodic instead of after every query evaluation, may be considered.

7.2.2 *Sophisticated History-aware Self-Adaptation*

We introduced a reference architecture for an adaptation engine which integrates InTempo with the MAPE-K loop and is thus capable of history-aware adaptation. The engine focused on the Analyze activity where adaptation issues are detected based on the knowledge base, i.e., temporal queries are evaluated over the runtime model with history, and details of other activities were abstracted. The performed adaptations were based on straightforward adaptation rules. In more sophisticated adaptation scenarios, the Analyze activity may have to consider more rules and thus perform multiple queries; based on the issues found, the Plan activity may have to perform its own analysis to compute the impact of the application of each rule, devise an adaptation plan, and ensure that the plan addresses adaptation issues in an optimal manner.

The duration of these tasks may be non-negligible. We plan to apply the engine in such sophisticated adaptation scenarios and investigate its performance. Moreover, engine variants can be developed based on features of InTempo: one variant may use non-definite answers for the early detection of possible issues, whereas another may be interested only in definite albeit slower answers; provided that timing constraints in the input queries allow it, another variant may temporarily enable the pruning option, if it observes that query evaluation speed does not satisfy the requirements of the adaptation.

An unexploited feature of the temporal validity returned by the querying approach is that, for certain properties, the query result can be combined with the non-definiteness window to compute the time remaining for the falsification of a temporal requirement. This knowledge may enable more informed decision-making during the Plan activity.

On a foundational level, a direction which holds vast potential is the opportunity to observe and learn from the history of the architecture and the performed adaptations. We plan to leverage the direct access to the history recorded in the model and the efficient querying approach to enable an adaptation engine with the capability for predictions.

7.2.3 *History-awareness and Temporal Queries for Systems of Systems*

Increasingly, systems operate in settings where they have to collaborate with other systems, thus forming systems of systems. From a model-driven engineering perspective, this collaboration can be realized by multiple runtime models in so-called megamodels. A significantly promising direction is work on the foundations and technology that will allow for history-awareness in megamodels, for example by handling synchronization and combination of multiple history-aware runtime models.

In this setting, the answers from local evaluations of temporal queries will have to be combined. This requirement poses significant yet fascinating challenges. Two areas of particular interest for our future work are methods for distributed evaluation for temporal queries and the handling of synchronization issues, e.g., network failures or out-of-order events, in query evaluations.

BIBLIOGRAPHY

[1] James F. Allen. “Maintaining Knowledge About Temporal Intervals.” In: *Communications of the ACM* 26.11 (Nov. 1983), pp. 832–843. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/182.358434.

[2] Bowen Alpern and Fred B. Schneider. “Recognizing Safety and Liveness.” In: *Distributed Computing* 2.3 (Sept. 1987), pp. 117–126. ISSN: 0178-2770, 1432-0452. DOI: 10.1007/BF01782772.

[3] Apache Foundation. *EventStrategy - Apache TinkerPop Reference Documentation*. URL: https://tinkerpop.apache.org/docs/3.5.1/reference/#_eventstrategy (visited on 07/25/2023).

[4] Kyungmin Bae and Jia Lee. “Bounded Model Checking of Signal Temporal Logic Properties Using Syntactic Separation.” In: *Proceedings of the ACM on Programming Languages* 3 (POPL Jan. 2, 2019), 51:1–51:30. DOI: 10.1145/3290364.

[5] Matthias Barkowsky and Holger Giese. “Hybrid Search Plan Generation for Generalized Graph Pattern Matching.” In: *Journal of Logical and Algebraic Methods in Programming* 114 (Aug. 1, 2020), p. 100563. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2020.100563.

[6] Matthias Barkowsky and Holger Giese. “Towards Development with Multi-version Models: Detecting Merge Conflicts and Checking Well-Formedness.” In: *Graph Transformation*. Ed. by Nicolas Behr and Daniel Strüber. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 118–136. ISBN: 978-3-031-09843-7. DOI: 10.1007/978-3-031-09843-7_7.

[7] Konstantinos Barmpis, Seyyed Shah, and Dimitrios S. Kolovos. “Towards Incremental Updates in Large-Scale Model Indexes.” In: *Modelling Foundations and Applications*. Ed. by Gabriele Taentzer and Francis Bordeleau. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 137–153. ISBN: 978-3-319-21151-0. DOI: 10.1007/978-3-319-21151-0_10.

[8] Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. “Extending Complex Event Processing to Graph-structured Information.” In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS ’18. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2018, pp. 166–175. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239402.

[9] Gala Barquero, Javier Troya, and Antonio Vallecillo. “Trading Accuracy for Performance in Data Processing Applications.” In: *The Journal of Object Technology* 18.2 (2019), 9:1. ISSN: 1660-1769. DOI: 10.5381/jot.2019.18.2.a9.

[10] Gala Barquero, Javier Troya, and Antonio Vallecillo. "Improving Query Performance on Dynamic Graphs." In: *Software and Systems Modeling* 20.4 (Aug. 1, 2021), pp. 1011–1041. ISSN: 1619-1374. DOI: 10.1007/s10270-020-00832-3.

[11] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. "Introduction to Runtime Verification." In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 1–33. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_1.

[12] Ezio Bartocci et al. "First International Competition on Runtime Verification: Rules, Benchmarks, Tools, and Final Results of CRV 2014." In: *International Journal on Software Tools for Technology Transfer* 21.1 (Feb. 1, 2019), pp. 31–70. ISSN: 1433-2787. DOI: 10.1007/s10009-017-0454-5.

[13] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. "Monitoring Metric First-Order Temporal Properties." In: *Journal of the ACM* 62.2 (May 6, 2015), 15:1–15:45. ISSN: 0004-5411. DOI: 10.1145/2699444.

[14] David Basin, Felix Klaedtke, and Eugen Zălinescu. "The MonPoly Monitoring Tool." In: *Kalpa Publications in Computing*. RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. Vol. 3. EasyChair, Dec. 14, 2017, pp. 19–28. DOI: 10.29007/89hs.

[15] David Basin, Felix Klaedtke, and Eugen Zălinescu. "Algorithms for Monitoring Real-Time Properties." In: *Acta Informatica* 55.4 (June 1, 2018), pp. 309–338. ISSN: 1432-0525. DOI: 10.1007/s00236-017-0295-4.

[16] Andreas Bauer, Martin Leucker, and Christian Schallhart. "The Good, the Bad, and the Ugly, But How Ugly Is Ugly?" In: *Runtime Verification*. Ed. by Oleg Sokolsky and Serdar Taşıran. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 126–138. ISBN: 978-3-540-77395-5. DOI: 10.1007/978-3-540-77395-5_11.

[17] Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. "Active Operations on Collections." In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 91–105. ISBN: 978-3-642-16145-2. DOI: 10.1007/978-3-642-16145-2_7.

[18] Nelly Bencomo, Sebastian Götz, and Hui Song. "Models@run.Time: A Guided Tour of the State of the Art and Research Challenges." In: *Software & Systems Modeling* 18.5 (Oct. 1, 2019), pp. 3049–3082. ISSN: 1619-1374. DOI: 10.1007/s10270-018-00712-x.

[19] Nelly Bencomo, Kris Welsh, Pete Sawyer, and Jon Whittle. "Self-Explanation in Adaptive Systems." In: *Proceedings of the 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. ICECCS '12. USA: IEEE Computer Society, July 18, 2012, pp. 157–166. ISBN: 978-2-9541810-0-4.

[20] Amine Benelallam, Thomas Hartmann, Ludovic Mouline, Francois Fouquet, Johann Bourcier, Olivier Barais, and Yves Le Traon. "Raising Time Awareness in Model-Driven Engineering: Vision Paper." In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2017, pp. 181–188. DOI: 10.1109/MODELS.2017.11.

[21] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. "Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems." In: *IEEE Transactions on Parallel and Distributed Systems* (2021), pp. 1–1. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2021.3131677.

[22] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. 2nd ed. Packt Publishing, 2016. 426 pp. ISBN: 978-1-78646-496-5.

[23] Thomas Beyhl, Dominique Blouin, Holger Giese, and Leen Lambers. "On the Operationalization of Graph Queries with Generalized Discrimination Networks." In: *Graph Transformation*. Ed. by Rachid Echahed and Mark Minas. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 170–186. ISBN: 978-3-319-40530-8. DOI: 10.1007/978-3-319-40530-8_11.

[24] Jean Bézivin, Richard F. Paige, Uwe Aßmann, Bernhard Rumpe, and Douglas C. Schmidt. "08331 Manifesto – Model Engineering for Complex Systems." In: *Perspectives Workshop: Model Engineering of Complex Systems (MECS)*. Ed. by Uwe Aßmann, Jean Bézivin, Richard Paige, Bernhard Rumpe, and Douglas C. Schmidt. Dagstuhl Seminar Proceedings 08331. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008. URL: http://drops.dagstuhl.de/opus/volltexte/2008/1603.

[25] Enrico Biermann, Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Gabriele Taentzer. "Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation." In: *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of His 65th Birthday*. Ed. by Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 121–140. ISBN: 978-3-642-17322-6. DOI: 10.1007/978-3-642-17322-6_7.

[26] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation." In: *Software & Systems Modeling* 11.2 (May 1, 2012), pp. 227–250. ISSN: 1619-1374. DOI: 10.1007/s10270-011-0199-7.

[27] Robert Bill, Alexandra Mazak, Manuel Wimmer, and Birgit Vogel-Heuser. "On the Need for Temporal Model Repositories." In: *Software Technologies: Applications and Foundations*. Ed. by Martina Seidl and Steffen Zschaler. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 136–145. ISBN: 978-3-319-74730-9. DOI: 10.1007/978-3-319-74730-9_11.

[28] Gordon Blair, Nelly Bencomo, and Robert B. France. "Models@ Run.Time." In: *Computer* 42.10 (Oct. 2009), pp. 22–27. ISSN: 1558-0814. DOI: 10.1109/MC.2009.326.

[29] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi Parache. "Trace-Checking Signal-Based Temporal Properties: A Model-Driven Approach." In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Virtual Event (Australia): ACM, Dec. 21, 2020, pp. 1004–1015. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3324884.3416631.

[30] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. "Model-Driven Software Engineering in Practice, Second Edition." In: *Synthesis Lectures on Software Engineering* 3.1 (Mar. 30, 2017), pp. 1–207. ISSN: 2328-3319. DOI: 10.2200/S00751ED2V01Y201701SWE004.

[31] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. "Grand Challenges in Model-Driven Engineering: An Analysis of the State of the Research." In: *Software and Systems Modeling* 19.1 (Jan. 1, 2020), pp. 5–13. ISSN: 1619-1374. DOI: 10.1007/s10270-019-00773-6.

[32] Márton Búr, Kristóf Marussy, Brett H. Meyer, and Dániel Varró. "Worst-Case Execution Time Calculation for Query-based Monitors by Witness Generation." In: *ACM Transactions on Embedded Computing Systems* 20.6 (Oct. 18, 2021), 107:1–107:36. ISSN: 1539-9087. DOI: 10.1145/3471904.

[33] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. "Distributed Graph Queries Over Models@run.Time for Runtime Monitoring of Cyber-Physical Systems." In: *International Journal on Software Tools for Technology Transfer* 22.1 (Feb. 1, 2020), pp. 79–102. ISSN: 1433-2787. DOI: 10.1007/s10009-019-00531-5.

[34] Sven Burmester, Holger Giese, Andreas Seibel, and Matthias Tichy. "Worst-Case Execution Time Optimization of Story Patterns for Hard Real-Time Systems." In: *Proc. of the 3rd International Fujaba Days*. 2005, pp. 71–78.

[35] Jordi Cabot, Antoni Olivé, and Ernest Teniente. "Representing Temporal Information in UML." In: *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Ed. by Perdita Stevens, Jon Whittle, and Grady Booch. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 44–59. ISBN: 978-3-540-45221-8. DOI: 10.1007/978-3-540-45221-8_5.

[36] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. "Kineograph: Taking the Pulse of a Fast-Changing and Connected World." In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. New York, NY, USA: Association for Computing Machinery, Apr. 10, 2012, pp. 85–98. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168846.

[37] Carlo Combi, Mauro Gambini, Sara Migliorini, and Roberto Posenato. "Modelling Temporal, Data-Centric Medical Processes." In: *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*. IHI '12. New York, NY, USA: Association for Computing Machinery, Jan. 28, 2012, pp. 141–150. ISBN: 978-1-4503-0781-9. DOI: 10.1145/2110363.2110382.

[38] Bruno Courcelle. "The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic." In: *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. USA: World Scientific Publishing Co., Inc., Feb. 1, 1997, pp. 313–400. ISBN: 978-981-02-2884-2.

[39] Gianpaolo Cugola and Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing." In: *ACM Computing Surveys* 44.3 (June 14, 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677.

[40] István Dávid, István Ráth, and Dániel Varró. "Foundations for Streaming Model Transformations by Complex Event Processing." In: *Software & Systems Modeling* 17.1 (Feb. 2018), pp. 135–162. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-016-0533-1.

[41] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. "A Model and Query Language for Temporal Graph Databases." In: *The VLDB Journal* 30.5 (Sept. 1, 2021), pp. 825–858. ISSN: 0949-877X. DOI: 10.1007/s00778-021-00675-4.

[42] Normann Decker, Franziska Kühn, and Daniel Thoma. "Runtime Verification of Web Services for Interconnected Medical Devices." In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014 IEEE 25th International Symposium on Software Reliability Engineering. Nov. 2014, pp. 235–244. DOI: 10.1109/ISSRE.2014.16.

[43] Rogério de Lemos et al. "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap." In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 1–32. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_1.

[44] Thomas J. DiCiccio and Bradley Efron. "Bootstrap Confidence Intervals." In: *Statistical Science* 11.3 (Sept. 1996), pp. 189–228. ISSN: 0883-4237, 2168-8745. DOI: 10.1214/ss/1032280214.

[45] Wei Dou, Domenico Bianculli, and Lionel Briand. “OCLR: A More Expressive, Pattern-Based Temporal Extension of OCL.” In: *Modelling Foundations and Applications*. Ed. by Jordi Cabot and Julia Rubin. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 51–66. ISBN: 978-3-319-09195-2. DOI: 10.1007/978-3-319-09195-2_4.

[46] Wei Dou, Domenico Bianculli, and Lionel Briand. “A Model-Driven Approach to Trace Checking of Pattern-Based Temporal Properties.” In: *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*. MODELS ’17. Austin, Texas: IEEE Press, Sept. 17, 2017, pp. 323–333. ISBN: 978-1-5386-3492-9. DOI: 10.1109/MODELS.2017.9.

[47] Wei Dou, Domenico Bianculli, and Lionel Briand. “TemPsy-Check: A Tool for Model-driven Trace Checking of Pattern-based Temporal Properties.” In: *Kalpa Publications in Computing*. RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. Vol. 3. EasyChair, Dec. 14, 2017, pp. 64–70. DOI: 10.29007/w2nj.

[48] Wei Dou, Domenico Bianculli, and Lionel Briand. “Model-Driven Trace Diagnostics for Pattern-based Temporal Specifications.” In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS ’18. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2018, pp. 278–288. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239396.

[49] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. “Patterns in Property Specifications for Finite-State Verification.” In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. May 1999, pp. 411–420. DOI: 10.1145/302405.302672.

[50] Eclipse Foundation. *Eclipse Hawk*. 2019. URL: https://www.eclipse.org/hawk/ (visited on 07/25/2023).

[51] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. URL: https://www.eclipse.org/modeling/emf/ (visited on 07/25/2023).

[52] Sebastian Ehmes, Lars Fritsche, and Konrad Altenhofen. “GrapeL: Combining Graph Pattern Matching and Complex Event Processing.” In: *Systems Modelling and Management*. Ed. by Önder Babur, Joachim Denil, and Birgit Vogel-Heuser. Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 180–196. ISBN: 978-3-030-58167-1. DOI: 10.1007/978-3-030-58167-1_13.

[53] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. “Fundamental Theory for Typed Attributed Graph Transformation.” In: *Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 161–177. ISBN: 978-3-540-30203-2. DOI: 10.1007/978-3-540-30203-2_13.

[54] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. “Reasoning with Temporal Logic on Truncated Paths.” In: *Computer Aided Verification*. Ed. by Warren A. Hunt and Fabio Somenzi. Berlin, Heidelberg: Springer, 2003, pp. 27–39. ISBN: 978-3-540-45069-6. DOI: 10.1007/978-3-540-45069-6_3.

[55] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. “The LDBC Social Network Benchmark: Interactive Workload.” In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. New York, NY, USA: Association for Computing Machinery, May 27, 2015, pp. 619–630. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742786.

[56] Naeem Esfahani, Eric Yuan, Kyle R. Canavera, and Sam Malek. “Inferring Software Component Interaction Dependencies for Adaptation Support.” In: *ACM Transactions on Autonomous and Adaptive Systems* 10.4 (Feb. 3, 2016), 26:1–26:32. ISSN: 1556-4665. DOI: 10.1145/2856035.

[57] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. “Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java.” In: *Theory and Application of Graph Transformations, 6th International Workshop, TAGT’98, Paderborn, Germany, November 16-20, 1998, Selected Papers*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 1764. Lecture Notes in Computer Science. Springer, 1998, pp. 296–309. DOI: 10.1007/978-3-540-46464-8_21.

[58] Charles L. Forgy. “Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.” In: *Artificial Intelligence* 19.1 (Sept. 1982), pp. 17–37. ISSN: 00043702. DOI: 10.1016/0004-3702(82)90020-0.

[59] Robert France and Bernhard Rumpe. “Model-Driven Development of Complex Software: A Research Roadmap.” In: *Future of Software Engineering (FOSE ’07)*. May 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[60] Antonio García-Domínguez, Nelly Bencomo, Juan Marcelo Parra Ullauri, and Luis Hernan Garcia Paucar. “Towards History-Aware Self-Adaptation with Explanation Capabilities.” In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. Umea, Sweden: IEEE, June 2019, pp. 18–23. ISBN: 978-1-72812-406-3. DOI: 10.1109/FAS-W.2019.00018.

[61] Antonio García-Domínguez, Nelly Bencomo, Juan Marcelo Parra-Ullauri, and Luis Hernán García-Paucar. “Querying and Annotating Model Histories with Time-Aware Patterns.” In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2019, pp. 194–204. DOI: 10.1109/MODELS.2019.000-2.

[62] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. "Software Architecture-Based Self-Adaptation." In: *Autonomic Computing and Networking*. Ed. by Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko. Boston, MA: Springer US, 2009, pp. 31–55. ISBN: 978-0-387-89827-8. DOI: 10.1007/978-0-387-89828-5_2.

[63] Sona Ghahremani, Holger Giese, and Thomas Vogel. "Improving Scalability and Reward of Utility-Driven Self-Healing for Large Dynamic Architectures." In: *ACM Transactions on Autonomous and Adaptive Systems* 14.3 (Feb. 25, 2020), 12:1–12:41. ISSN: 1556-4665. DOI: 10.1145/3380965.

[64] Holger Giese, Nelly Bencomo, Liliana Pasquale, Andres J. Ramirez, Paola Inverardi, Sebastian Wätzoldt, and Siobhán Clarke. "Living with Uncertainty in the Age of Runtime Models." In: *Models@run.Time: Foundations, Applications, and Roadmaps*. Ed. by Nelly Bencomo, Robert France, Betty H. C. Cheng, and Uwe Aßmann. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 47–100. ISBN: 978-3-319-08915-7. DOI: 10.1007/978-3-319-08915-7_3.

[65] Holger Giese, Stefan Henkler, Martin Hirsch, Florian Klein, and Michael Spijkerman. "Monitoring of Structural and Temporal Properties." In: *Proc. of the 5th International Fujaba Days 2007, Kassel, Germany*. Ed. by Leif Geiger, Holger Giese, and Albert Zündorf. Sept. 2007, pp. 8–11.

[66] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. "Improved Flexibility and Scalability by Interpreting Story Diagrams." In: *Electronic Communications of the EASST* 18.0 (Sept. 8, 2009). ISSN: 1863-2122. DOI: 10.14279/tuj.eceasst.18.268.

[67] Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt. "Graph Transformations for MDE, Adaptation, and Models at Runtime." In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 137–191. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_5.

[68] Holger Giese, Maria Maximova, Lucas Sakizloglou, and Sven Schneider. "Metric Temporal Graph Logic over Typed Attributed Graphs." In: *Fundamental Approaches to Software Engineering*. Ed. by Reiner Hähnle and Wil van der Aalst. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 282–298. ISBN: 978-3-030-16722-6. DOI: 10.1007/978-3-030-16722-6_16.

[69] Ulrike Golas, Annegret Habel, and Hartmut Ehrig. "Multi-Amalgamation of Rules with Application Conditions in -Adhesive Categories." In: *Mathematical Structures in Computer Science* 24.4 (Aug. 2014). ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129512000345.

[70] Heather J. Goldsby, Betty H. C. Cheng, and Ji Zhang. "AMOEBA-RT: Run-Time Verification of Adaptive Software." In: *Models in Software Engineering*. Ed. by Holger Giese. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 212–224. ISBN: 978-3-540-69073-3. DOI: 10.1007/978-3-540-69073-3_23.

[71] Abel Gómez, Jordi Cabot, and Manuel Wimmer. "TemporalEMF: A Temporal Metamodeling Framework." In: *Conceptual Modeling*. Ed. by Juan C. Trujillo, Karen C. Davis, Xiaoyong Du, Zhanhuai Li, Tok Wang Ling, Guoliang Li, and Mong Li Lee. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 365–381. ISBN: 978-3-030-00847-5. DOI: 10.1007/978-3-030-00847-5_26.

[72] Google Inc. *Google Core Libraries for Java*. Guava. URL: https://github.com/google/guava (visited on 07/25/2023).

[73] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. "On the Expressiveness of Languages for Complex Event Recognition." In: *23rd International Conference on Database Theory (ICDT 2020)*. Ed. by Carsten Lutz and Jean Christoph Jung. Vol. 155. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 15:1–15:17. ISBN: 978-3-95977-139-9. DOI: 10.4230/LIPIcs.ICDT.2020.15.

[74] Szilvia Gyapay, Reiko Heckel, and Dániel Varró. "Graph Transformation with Time: Causality and Logical Clocks." In: *Graph Transformation*. Ed. by Andrea Corradini, Hartmut Ehrig, Hans -Jörg Kreowski, and Grzegorz Rozenberg. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2505. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 120–134. ISBN: 978-3-540-45832-6. DOI: 10.1007/3-540-45832-8_11.

[75] Annegret Habel and Karl-Heinz Pennemann. "Correctness of High-Level Transformation Systems Relative to Nested Conditions." In: *Mathematical Structures in Computer Science* 19.2 (Apr. 1, 2009), pp. 245–296. ISSN: 0960-1295.

[76] Martin Haeusler, Thomas Trojer, Johannes Kessler, Matthias Farwick, Emmanuel Nowakowski, and Ruth Breu. "Chronosphere: A Graph-Based EMF Model Repository for IT Landscape Models." In: *Software and Systems Modeling* 18.6 (Dec. 1, 2019), pp. 3487–3526. ISSN: 1619-1374. DOI: 10.1007/s10270-019-00725-0.

[77] Sylvain Hallé. "When RV Meets CEP." In: *Runtime Verification*. Ed. by Yliès Falcone and César Sánchez. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 68–91. ISBN: 978-3-319-46982-9. DOI: 10.1007/978-3-319-46982-9_6.

[78] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. "Chronos: A Graph Engine for Temporal Graph Analysis." In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. New York, NY, USA: ACM, 2014, 1:1–1:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592799.

[79] E. N. Hanson, S. Bodagala, and U. Chadaga. “Trigger Condition Testing and View Maintenance Using Optimized Discrimination Networks.” In: *IEEE Transactions on Knowledge and Data Engineering* 14.2 (Mar. 2002), pp. 261–280. ISSN: 1558-2191. DOI: 10.1109/69.991716.

[80] Thomas Hartmann, François Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. “Analyzing Complex Data in Motion at Scale with Temporal Graphs.” In: *The 29th International Conference on Software Engineering and Knowledge Engineering, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 5-7, 2017*. Ed. by Xudong He. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2017, pp. 596–601. DOI: 10.18293/SEKE2017-048.

[81] Klaus Havelund, Moran Omer, and Doron Peled. “Monitoring First-Order Interval Logic.” In: *Software Engineering and Formal Methods*. Ed. by Radu Calinescu and Corina S. Păsăreanu. Cham: Springer International Publishing, 2021, pp. 66–83. ISBN: 978-3-030-92124-8. DOI: 10.1007/978-3-030-92124-8_4.

[82] Klaus Havelund and Doron Peled. “Efficient Runtime Verification of First-Order Temporal Properties.” In: *Model Checking Software*. Ed. by María del Mar Gallardo and Pedro Merino. Vol. 10869. Cham: Springer International Publishing, 2018, pp. 26–47. ISBN: 978-3-319-94110-3.

[83] Klaus Havelund and Doron Peled. “Runtime Verification: From Propositional to First-Order Temporal Logic.” In: *Runtime Verification*. Ed. by Christian Colombo and Martin Leucker. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 90–112. ISBN: 978-3-030-03769-7.

[84] Klaus Havelund and Doron Peled. “BDDs for Representing Data in Runtime Verification.” In: *Runtime Verification*. Ed. by Jyotirmoy Deshmukh and Dejan Ničković. Vol. 12399. Cham: Springer International Publishing, 2020, pp. 107–128. ISBN: 978-3-030-60508-7. DOI: 10.1007/978-3-030-60508-7_6.

[85] Klaus Havelund and Doron Peled. “First-Order Timed Runtime Verification Using BDDs.” In: *Automated Technology for Verification and Analysis*. Ed. by Dang Van Hung and Oleg Sokolsky. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 3–24. ISBN: 978-3-030-59152-6. DOI: 10.1007/978-3-030-59152-6_1.

[86] Klaus Havelund and Doron Peled. “An Extension of First-Order LTL with Rules with Application to Runtime Verification.” In: *International Journal on Software Tools for Technology Transfer* 23.4 (Aug. 1, 2021), pp. 547–563. ISSN: 1433-2787. DOI: 10.1007/s10009-021-00626-y.

[87] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. “Monitoring Events That Carry Data.” In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 61–102. ISBN: 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_3.

[88] Stephan Hildebrandt. "On the Performance and Conformance of Triple Graph Grammar Implementations." PhD thesis. Germany: Hasso Plattner Institute, University of Potsdam, June 2014.

[89] Nicolas Hili, Mojtaba Bagherzadeh, Karim Jahed, and Juergen Dingel. "A Model-Based Architecture for Interactive Run-Time Monitoring." In: *Software and Systems Modeling (SoSyM)* 19.4 (July 1, 2020), pp. 959–981. ISSN: 1619-1366. DOI: 10.1007/s10270-020-00780-y.

[90] Paul Hunter, Joel Ouaknine, and James Worrell. "Expressive Completeness for Metric Temporal Logic." In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '13. USA: IEEE Computer Society, June 25, 2013, pp. 349–357. ISBN: 978-0-7695-5020-6. DOI: 10.1109/LICS.2013.41.

[91] Didac Gil De La Iglesia and Danny Weyns. "MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems." In: *ACM Transactions on Autonomous and Adaptive Systems* 10.3 (Sept. 1, 2015), 15:1–15:31. ISSN: 1556-4665. DOI: 10.1145/2724719.

[92] Bilal Kanso and Safouan Taha. "Specification of Temporal Properties with OCL." In: *Science of Computer Programming* 96.P4 (Dec. 15, 2014), pp. 527–551. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.02.029.

[93] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing." In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[94] Udayan Khurana and Amol Deshpande. "Efficient Snapshot Retrieval over Historical Graph Data." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. Apr. 2013, pp. 997–1008. DOI: 10.1109/ICDE.2013.6544892.

[95] Florian Klein and Holger Giese. "Joint Structural and Temporal Property Specification Using Timed Story Scenario Diagrams." In: *Fundamental Approaches to Software Engineering*. Ed. by Matthew B. Dwyer and Antónia Lopes. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 185–199. ISBN: 978-3-540-71289-3. DOI: 10.1007/978-3-540-71289-3_16.

[96] Ron Koymans. "Specifying Real-Time Properties with Metric Temporal Logic." In: *Real-Time Systems* 2.4 (Nov. 1, 1990), pp. 255–299. ISSN: 1573-1383. DOI: 10.1007/BF01995674.

[97] Christian Krause, Daniel Johannsen, Radwan Deeb, Kai-Uwe Sattler, David Knacker, and Anton Niadzelka. "An SQL-Based Query Language and Engine for Graph Pattern Matching." In: *Graph Transformation*. Ed. by Rachid Echahed and Mark Minas. Vol. 9761. Cham: Springer International Publishing, 2016, pp. 153–169. ISBN: 978-3-319-40530-8. DOI: 10.1007/978-3-319-40530-8_10.

[98] Pradeep Kumar and H. Howie Huang. "GraphOne: A Data Store for Real-time Analytics on Evolving Graphs." In: *ACM Transactions on Storage* 15.4 (Jan. 16, 2020), 29:1–29:40. ISSN: 1553-3077. DOI: 10.1145/3364180.

[99] Orna Kupferman and Moshe Y. Vardi. "Model Checking of Safety Properties." In: *Formal Methods in System Design* 19.3 (Nov. 1, 2001), pp. 291–314. ISSN: 1572-8102. DOI: 10.1023/A:1011254632723.

[100] Leslie Lamport. "Proving the Correctness of Multiprocess Programs." In: *IEEE Transactions on Software Engineering* SE-3.2 (Mar. 1977), pp. 125–143. ISSN: 1939-3520. DOI: 10.1109/TSE.1977.229904.

[101] Ivan Lanese, Antonio Bucchiarone, and Fabrizio Montesi. "A Framework for Rule-Based Dynamic Adaptation." In: *Trustworthly Global Computing*. Ed. by Martin Wirsing, Martin Hofmann, and Axel Rauschmayer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 284–300. ISBN: 978-3-642-15640-3. DOI: 10.1007/978-3-642-15640-3_19.

[102] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. "Temporal Logic with Forgettable Past." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2002, pp. 383–392.

[103] LDBC Social Network Benchmark task force. *The LDBC Social Network Benchmark (Version 2.2.1)*. Linked Data Benchmark Council, 2022. URL: https://github.com/ldbc/ldbc_snb_docs.

[104] Pattie Maes. "Concepts and Experiments in Computational Reflection." In: *ACM SIGPLAN Notices* 22.12 (Dec. 1, 1987), pp. 147–155. ISSN: 0362-1340. DOI: 10.1145/38807.38821.

[105] Jeff Magee and Jeff Kramer. "Dynamic Structure in Software Architectures." In: *ACM SIGSOFT Software Engineering Notes* 21.6 (Oct. 1, 1996), pp. 3–14. ISSN: 0163-5948. DOI: 10.1145/250707.239104.

[106] Oded Maler, Dejan Nickovic, and Amir Pnueli. "Checking Temporal Properties of Discrete, Timed and Continuous Behaviors." In: *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Ed. by Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 475–505. ISBN: 978-3-540-78127-1. DOI: 10.1007/978-3-540-78127-1_26.

[107] Oded Maler and Dejan Ničković. "Monitoring Properties of Analog and Mixed-Signal Circuits." In: *International Journal on Software Tools for Technology Transfer* 15.3 (June 1, 2013), pp. 247–268. ISSN: 1433-2787. DOI: 10.1007/s10009-012-0247-9.

[108] Felix Mannhardt and Daan Blinde. "Analyzing the Trajectories of Patients with Sepsis Using Process Mining." In: *RADAR+EMISA@CAiSE, Essen, Germany, June 12-13, 2017*. Ed. by Jens Gulden, Selmin Nurcan, Iris Reinhartz-Berger, Wided Guédria, Palash Bera, Sérgio Guerreiro, Michael Fellmann, and Matthias Weidlich. Vol. 1859. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 72–80. URL: http://ceur-ws.org/Vol-1859/bpmds-08-paper.pdf.

[109] Diego Marmsoler and Ana Petrovska. "Runtime Verification for Dynamic Architectures." In: *Journal of Logical and Algebraic Methods in Programming* 118 (Jan. 1, 2021), p. 100618. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2020.100618.

[110] Maria Massri, Philippe Raipin, and Pierre Meye. “GDBAlive: A Temporal Graph Database Built on Top of a Columnar Data Store.” In: *Journal of Advances in Information Technology* 12.3 (2021). ISSN: 17982340. DOI: 10.12720/jait.12.3.169-178.

[111] Alexandra Mazak, Sabine Wolny, Abel Gómez, Jordi Cabot, Manuel Wimmer, and Gerti Kappel. “Temporal Models on Time Series Databases.” In: *The Journal of Object Technology* 19.3 (2020), 3:1. ISSN: 1660-1769. DOI: 10.5381/jot.2020.19.3.a14.

[112] Philip K. McKinley, Sayed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. “Composing Adaptive Software.” In: *Computer* 37.7 (July 2004), pp. 56–64. ISSN: 1558-0814. DOI: 10.1109/MC.2004.48.

[113] MDELab. *InTempo Eclipse Plugin Homepage*. 2020. URL: https://www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/intempo/ (visited on 07/25/2023).

[114] MDELab. *SDM Tools*. URL: https://www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/story-diagram-tools/ (visited on 07/25/2023).

[115] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. “ImmortalGraph: A System for Storage and Analysis of Temporal Graphs.” In: *ACM Transactions on Storage* 11.3 (July 24, 2015), 14:1–14:34. ISSN: 1553-3077. DOI: 10.1145/2700302.

[116] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Jan. 2009. ISBN: 978-0-89871-771-6. DOI: 10.1137/1.9780898717716.

[117] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. “Models@ Run.Time to Support Dynamic Adaptation.” In: *Computer* 42.10 (Oct. 1, 2009), pp. 44–51. ISSN: 0018-9162. DOI: 10.1109/MC.2009.327.

[118] United Kingdom National Health Service. *Records Management Code of Practice for Health and Social Care 2021*. 2021. URL: https://transform.england.nhs.uk/information-governance/guidance/records-management-code/ (visited on 07/25/2023).

[119] Office for Civil Rights (OCR). *Summary of the HIPAA Privacy Rule*. HHS.gov. May 7, 2008. URL: https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html (visited on 07/25/2023).

[120] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. “An Architecture-Based Approach to Self-Adaptive Software.” In: *IEEE Intelligent Systems* 14.3 (May 1, 1999), pp. 54–62. ISSN: 1541-1672. DOI: 10.1109/5254.769885.

[121] Juan Marcelo Parra-Ullauri, Antonio García-Domínguez, Nelly Bencomo, Changgang Zheng, Chen Zhen, Juan Boubeta-Puig, Guadalupe Ortiz, and Shufan Yang. "Event-Driven Temporal Models for Explanations - ETeMoX: Explaining Reinforcement Learning." In: *Software and Systems Modeling* 21.3 (June 1, 2022), pp. 1091–1113. ISSN: 1619-1374. DOI: 10.1007/s10270-021-00952-4.

[122] Doron Peled and Klaus Havelund. "Refining the Safety–Liveness Classification of Temporal Properties According to Monitorability." In: *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*. Ed. by Tiziana Margaria, Susanne Graf, and Kim G. Larsen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 218–234. ISBN: 978-3-030-22348-9. URL: https://doi.org/10.1007/978-3-030-22348-9_14.

[123] Giles Reger and David Rydeheard. "From First-order Temporal Logic to Parametric Trace Slicing." In: *Runtime Verification*. Ed. by Ezio Bartocci and Rupak Majumdar. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 216–232. ISBN: 978-3-319-23820-3. DOI: 10.1007/978-3-319-23820-3_14.

[124] Arend Rensink. "Representing First-Order Logic Using Graphs." In: *Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 319–335. ISBN: 978-3-540-30203-2. DOI: 10.1007/978-3-540-30203-2_23.

[125] Owen Reynolds, Antonio García-Domínguez, and Nelly Bencomo. "Cronista: A Multi-Database Automated Provenance Collection System for Runtime-Models." In: *Information and Software Technology* 141 (Jan. 1, 2022), p. 106694. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106694.

[126] Andrew Rhodes et al. "Surviving Sepsis Campaign: International Guidelines for Management of Sepsis and Septic Shock: 2016." In: *Intensive Care Medicine* 43.3 (Mar. 1, 2017), pp. 304–377. ISSN: 1432-1238. DOI: 10.1007/s00134-017-4683-6.

[127] Mark Richters and Martin Gogolla. "OCL: Syntax, Semantics, and Tools." In: *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. Ed. by Tony Clark and Jos Warmer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 42–68. ISBN: 978-3-540-45669-8. DOI: 10.1007/3-540-45669-4_4.

[128] William Robinson. "Extended OCL for Goal Monitoring." In: *Electronic Communications of the EASST* 9.0 (0 Nov. 23, 2007). ISSN: 1863-2122. DOI: 10.14279/tuj.eceasst.9.105.

[129] Tobias Rötschke and Andy Schürr. "Temporal Graph Queries to Support Software Evolution." In: *Graph Transformations*. Ed. by Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 291–305. ISBN: 978-3-540-38872-2. DOI: 10.1007/11841883_21.

[130] Patrice C Roy, Samina Raza Abidi, and Syed Sibte Raza Abidi. "Monitoring Medication Adherence in Smart Environments in the Context of Patient Self-Management: A Knowledge-Driven Approach." In: *Smart Technologies in Healthcare*. CRC Press, 2017, pp. 195–223.

[131] Eric Rutten, Nicolas Marchand, and Daniel Simon. "Feedback Control as MAPE-K Loop in Autonomic Computing." In: *Software Engineering for Self-Adaptive Systems III. Assurances*. Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 349–373. ISBN: 978-3-319-74183-3. DOI: 10.1007/978-3-319-74183-3_12.

[133] Lucas Sakizloglou and Sona Ghahremani. *Simplified Event Logs for Sepsis Patient Trajectories*. 2020. URL: https://zenodo.org/record/3989590 (visited on 07/25/2023).

[134] Lucas Sakizloglou, Sona Ghahremani, and Matthias Barkowsky. *Event Logs Containing Deletions*. 2021. URL: http://doi.org/10.5281/zenodo.5042439 (visited on 07/25/2023).

[138] Mazeiar Salehie and Ladan Tahvildari. "Self-Adaptive Software: Landscape and Research Challenges." In: *ACM Transactions on Autonomous and Adaptive Systems* 4.2 (May 21, 2009), 14:1–14:42. ISSN: 1556-4665. DOI: 10.1145/1516533.1516538.

[139] Jesús Sánchez Cuadrado and Juan de Lara. "Streaming Model Transformations: Scenarios, Challenges and Initial Solutions." In: *Theory and Practice of Model Transformations*. Ed. by Keith Duddy and Gerti Kappel. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 1–16. ISBN: 978-3-642-38883-5. DOI: 10.1007/978-3-642-38883-5_1.

[140] SAP OrientDB. *Live Query · OrientDB Manual*. URL: http://orientdb.com/docs/3.1.x/java/Live-Query.html (visited on 07/25/2023).

[141] Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. "A Formally Verified Monitor for Metric First-Order Temporal Logic." In: *Runtime Verification*. Ed. by Bernd Finkbeiner and Leonardo Mariani. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 310–328. ISBN: 978-3-030-32079-9. DOI: 10.1007/978-3-030-32079-9_18.

[142] Sven Schneider, Leen Lambers, and Fernando Orejas. "Automated Reasoning for Attributed Graph Properties." In: *International Journal on Software Tools for Technology Transfer* 20.6 (6 Nov. 1, 2018), pp. 705–737. ISSN: 1433-2787. DOI: 10.1007/s10009-018-0496-3.

[143] Sven Schneider, Maria Maximova, Lucas Sakizloglou, and Holger Giese. "Formal Testing of Timed Graph Transformation Systems Using Metric Temporal Graph Logic." In: *International Journal on Software Tools for Technology Transfer* 23.3 (June 2021), pp. 411–488. ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-020-00585-w.

[144] Sven Schneider, Lucas Sakizloglou, Maria Maximova, and Holger Giese. "Optimistic and Pessimistic On-the-fly Analysis for Metric Temporal Graph Logic." In: *Graph Transformation*. Ed. by Fabio Gadducci and Timo Kehrer. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 276–294. ISBN: 978-3-030-51372-6. DOI: 10.1007/978-3-030-51372-6_16.

[145] Michael Soden and Hajo Eichler. "Temporal Extensions of OCL Revisited." In: *Model Driven Architecture - Foundations and Applications*. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 190–205. ISBN: 978-3-642-02674-4. DOI: 10.1007/978-3-642-02674-4_14.

[146] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. "Event Pattern Matching over Graph Streams." In: *Proceedings of the VLDB Endowment* 8.4 (Dec. 1, 2014), pp. 413–424. ISSN: 2150-8097. DOI: 10.14778/2735496.2735504.

[147] Michael Spijkerman. "Monitoring Gemischt Struktureller und Temporaler Eigenschaften Von UML Modellen." MA thesis. Germany: Faculty of Software Engineering, University of Paderborn, Oct. 2007.

[148] Florian Stallmann. "A Model-Driven Approach to Multi-Agent System Design." PhD thesis. Germany: University of Paderborn, 2008. URL: https://digital.ub.uni-paderborn.de/hsmig/content/titleinfo/1109.

[149] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd ed. Addison-Wesley Professional, 2009. ISBN: 978-0-321-33188-5.

[150] Lars Stockmann, Sven Laux, and Eric Bodden. "Using Architectural Runtime Verification for Offline Data Analysis." In: *Journal of Automotive Software Engineering* 2.1 (2021), p. 1. ISSN: 2589-2258. DOI: 10.2991/jase.d.210205.001.

[151] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. "Variability-Based Model Transformation: Formal Foundation and Application." In: *Formal Aspects of Computing* 30.1 (Jan. 1, 2018), pp. 133–162. ISSN: 1433-299X. DOI: 10.1007/s00165-017-0441-3.

[152] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. "IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud." In: *Model-Driven Engineering Languages and Systems*. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 653–669. ISBN: 978-3-319-11653-2. DOI: 10.1007/978-3-319-11653-2_40.

[153] Michael Szvetits and Uwe Zdun. "Systematic Literature Review of the Objectives, Techniques, Kinds, and Architectures of Models at Runtime." In: *Software & Systems Modeling* 15.1 (Feb. 1, 2016), pp. 31–69. ISSN: 1619-1374. DOI: 10.1007/s10270-013-0394-9.

[154] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. "EMF-IncQuery: An Integrated Development Environment for Live Model Queries." In: *Science of Computer Programming* 98 (Feb. 1, 2015), pp. 80–99. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.01.004.

[155] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. "Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the Viatra Framework." In: *Software & Systems Modeling* 15.3 (July 1, 2016), pp. 609–629. ISSN: 1619-1374. DOI: 10.1007/s10270-016-0530-4.

[156] Thomas Vogel and Holger Giese. "Adaptation and Abstract Runtime Models." In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. New York, NY, USA: Association for Computing Machinery, May 3, 2010, pp. 39–48. ISBN: 978-1-60558-971-8. DOI: 10.1145/1808984.1808989.

[157] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. "Incremental Model Synchronization for Efficient Run-Time Monitoring." In: *Models in Software Engineering*. Ed. by Sudipto Ghosh. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 124–139. ISBN: 978-3-642-12261-3. DOI: 10.1007/978-3-642-12261-3_13.

[158] Thomas Vogel, Andreas Seibel, and Holger Giese. "The Role of Models and Megamodels at Runtime." In: *Models in Software Engineering*. Ed. by Juergen Dingel and Arnor Solberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 224–238. ISBN: 978-3-642-21210-9. DOI: 10.1007/978-3-642-21210-9_22.

[159] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 978-0-201-37940-2.

[160] Jack Waudby, Benjamin A. Steer, Arnau Prat-Pérez, and Gábor Szárnyas. "Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator." In: *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. SIGMOD/PODS '20: International Conference on Management of Data. Portland OR USA: ACM, June 14, 2020, pp. 1–8. ISBN: 978-1-4503-8021-8. DOI: 10.1145/3398682.3399165.

[161] Danny Weyns and Radu Calinescu. "Tele Assistance: A Self-Adaptive Service-Based System Examplar." In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '15. Florence, Italy: IEEE Press, May 16, 2015, pp. 88–92.

[162] Walter Wilson, Kathryn A. Taubert, Michael Gewitz, Peter B. Lockhart, Larry M. Baddour, Matthew Levison, Ann Bolger, Christopher H. Cabell, Masato Takahashi, Robert S. Baltimore, et al. "Prevention of Infective Endocarditis: A Guideline From the American Heart Associa-

tion." In: *Circulation* 116.15 (2007), pp. 1736–1754. URL: http://circ.ahajournals.org/content/116/15/1736.short.

[163] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Berlin Heidelberg: Springer-Verlag, 2012. ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2.

[164] Ji Zhang and Betty H. C. Cheng. "Using Temporal Logic to Specify Adaptive Program Semantics." In: *Journal of Systems and Software*. Architecting Dependable Systems 79.10 (Oct. 1, 2006), pp. 1361–1369. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.02.062.

[165] Paul Ziemann and Martin Gogolla. "OCL Extended with Temporal Logic." In: *Perspectives of System Informatics*. Ed. by Manfred Broy and Alexandre V. Zamulin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 351–357. ISBN: 978-3-540-39866-0. DOI: 10.1007/978-3-540-39866-0_35.

[166] Sheila Zingg, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. "Verified First-Order Monitoring with Recursive Rules." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 236–253. ISBN: 978-3-030-99527-0. DOI: 10.1007/978-3-030-99527-0_13.

[167] Albert Zündorf. "Rigorous Object Oriented Software Development." Habilitation Thesis. Germany: University of Paderborn, 2001.

A TECHNICAL SUPPLEMENT

A.1 INTERVALS

An *interval* is a non-empty, convex set $I \subseteq \mathbb{R}$ and has one of the following forms: $(\tau,\tau'), [\tau,\tau'), (\tau,\tau'], [\tau,\tau']$ where (i) τ,τ' is the *left* and *right* end-point of the interval, respectively (ii) $\tau \leq \tau'$ (iii) $\tau,\tau' \in \mathbb{R}$. A round bracket, square bracket denotes that the end-point is included in, respectively excluded from, the interval—correspondingly, an interval is *open* if both brackets are round, *half-open* if one of the brackets is round, or *closed* if both brackets are square. We also denote the left and right end-point of I by $\ell(I)$ and $r(I)$, respectively. For example, the interval $I = \{t \mid t \in \mathbb{R} \wedge 0 \leq t < 5\}$, denoted by $[0,5)$, is bounded and half-open (more precisely, left-closed and right-open) with $\ell(I) = 0$ and $r(I) = 5$. An interval is *bounded*, unless $r(I) = \infty$, whereby an interval is *unbounded*.

The set of all intervals is denoted by $\mathcal{I}$. Two intervals $k,l \in \mathcal{I}$ are *left-adjacent* when $r(k) = \ell(l)$ and k is right-open and l is left-closed; right-adjacent intervals are defined symmetrically. The two intervals are *overlapping* when $k \cap l \neq \varnothing$. For k, we denote the union $\ell(k) \cup k$, i.e., making k left-closed, by ${}^{+}k$ and the union $r(k) \cup k$ by k^{+}. Note that when $r(k) = \infty$, $k^{+} = k$.

We define the interval addition $\oplus$ and interval subtraction $\ominus$ as usual [116]. Let k,l be intervals in $\mathcal{I}$. Then,

$$k \oplus l = \{\tau + \tau' \mid \tau \in k, \tau' \in l\}$$

and

$$k \ominus l = \{\tau - \tau' \mid \tau \in k, \tau' \in l\}$$

respectively. Essentially,

$$k \oplus l = [\ell(k) + \ell(l), r(k) + r(l)]$$

and

$$k \ominus l = [\ell(k) - r(l), r(k) - \ell(l)]$$

Note the reversal of end-points in subtraction. In both operations the involvement of an open end-point leads to the end-point of the result also being open.

For the proof of Theorem 3.2.1, we rely on the distributivity of an interval addition or subtraction with the regular set intersection, captured by the lemma below.

Lemma A.1.1 (Distributivity of interval operation with regular set intersection)**.** *Let $K,L,I \in \mathcal{I}$ and $\star \in \{\oplus,\ominus\}$. If $K \cap L \neq \varnothing$, $(K \star I) \cap (L \star I) = (K \cap L) \star I$.*

Proof. Note that the negation $-I$ for the interval I is defined as $\{-\tau \mid \tau \in I\}$. It follows from the definition of addition and subtraction that $K \ominus I = K \oplus (-I)$ [see 116, p. 12], thus we only need to show the lemma for addition. The proof proceeds by showing inclusion in both directions.

We commence by hypothesizing $(K \oplus I) \cap (L \oplus I) \subseteq (K \cap L) \oplus I$. Let $\tau \in (K \oplus I) \cap (L \oplus I)$. It follows that $\tau \in (K \oplus I)$ and $\tau \in (L \oplus I)$. From these two memberships of τ, it follows that (i) there exists a $k \in K$ and an $i \in I$ such that $k + i = \tau$ (ii) there exists an $l \in L$ and an $i' \in I$ such that $l + i' = \tau$. Note that, since $K \cap L \neq \emptyset$, it may be that $k \in L$—then from $k \in L$, $k \in K$, and $k + i = \tau$ we can deduce $\tau \in (K \cap L) \oplus I$ and, in turn, that the initial hypothesis holds. A similar deduction can be made for the case when $l \in K$. In the case where $k \notin L$ and $l \notin K$, it follows that $k \neq l$. Assume, without loss of generality, that $k < l$. From the membership of τ it follows that there exists $z \in K \cap L$ with $k < z < l$ and a $t \in I$ with $i < t < i'$ such that $z + t = \tau$. From $z \in K$, $z \in L$, and $t \in I$, we can deduce that $z + t \in (K \cap L) \oplus I$. By $z + t = \tau$, we obtain $\tau \in (K \cap L) \oplus I$.

We proceed with inclusion $\supseteq$. Let $\tau \in (K \cap L) \oplus I$. It follows that there exists $z \in (K \cap L) \oplus I$ and an $i \in I$ such that $z + i = \tau$. From $z \in K, i \in I$ it follows that $\tau \in K \oplus I$ and, similarly, from $z \in L, i \in I$, it follows that $\tau \in L \oplus I$. Therefore $\tau \in (K \oplus I) \cap (L \oplus I)$.

By showing inclusion in both directions, we have shown the lemma to be true. □

A.2 AMALGAMATED GRAPH TRANSFORMATION RULES

In order to support the varying behavior of marking rules required by a TGDN, we employ amalgamated GT rules [25, 69] to realize *Amalgamated Marking Rules* (AMRs). This section elaborates on amalgamated GT rules based on the definitions in [25], focusing on non-deleting rules that feature neither ACs nor attributes—we refer to [69] for a comprehensive presentation also covering these aspects.

Amalgamated GT rules allow for matching various graph structures and *amalgamating*, i.e., synchronizing, these matches over a common kernel match. An amalgamated rule is based on a *kernel rule* and *multi-rules*. The kernel rule contains elements common to all rules and is to be applied only once; a multi-rule extends the kernel rule and may be applied arbitrarily many times. A *kernel morphism* embeds a kernel rule into a given multi-rule and is defined as follows: Given a rule $r_k := \langle f_k : n_k \hookrightarrow R_k \rangle$, called a kernel rule, and a rule $r_\mu := \langle f_\mu : n_\mu \hookrightarrow R_\mu \rangle$, called a multi-rule, a kernel morphism $s : r_k \rightarrow r_\mu, s = (s_n, s_R)$ consists of the monomorphisms $s_n : n_k \hookrightarrow n_\mu$ and $s_R : R_k \hookrightarrow R_\mu$ such that n_k contains only those elements that are contained in both n_μ and R_k and $s_R \circ f_k = s_n \circ f_\mu$.

Technically, an amalgamated GT rule is defined based on an *interaction scheme*, i.e., a bundle of kernel morphisms which describe how multi-rules relate to the kernel rule. An amalgamated rule is constructed via an application of the interaction scheme on a host graph: For each match for a multi-rule r_μ in the host graph which overlaps in a match for the kernel rule r_k, a copy of the multi-rule $r_{\mu,c}$ is created together with a kernel morphism $s_{\mu,c} : r_k \rightarrow r_{\mu,c}$ from the kernel rule to the copy; all copies are subsequently *glued* at the match of their common kernel rule which induces the amalgamated GT rule $\tilde{r}$. The

LHS of $\tilde{r}$ consists of all the LHSs of the multi-rule copies and the LHS of r_k; the RHS of $\tilde{r}$ consists of all the RHSs of the multi-rule copies and the RHS of r_k. When constructed, $\tilde{r}$ is a conventional GT rule.

The AMRs used in Section 4.1 are defined based on a single multi-rule. Therefore, the interaction scheme of an AMR is effectively the kernel morphism from the kernel rule to that multi-rule. The multi-rule of an AMR is an identity, i.e., its LHS is identical to its RHS, and only tracks marking nodes created by dependencies, such that the duration of these marking nodes can be taken into consideration by the kernel rule.

A.3 PROOFS

A.3.1 *Theorem 3.2.1: equality of satisfaction span and satisfaction computation*

Additionally to a set of convex sets of time points, i.e., intervals, a fragmented interval can be seen simply as a set of time points. In the following, when we utilize the latter view, we underline the fragmented interval, e.g., $\underline{\mathcal{Z}}$. Following is the proof for Theorem 3.2.1, i.e., given a match m in $H_{[\tau]}$ and an MTGC ψ, the satisfaction span $\mathcal{Y}$ of m for ψ over $H_{[c]}$ is given by the satisfaction computation $\mathcal{Z}$ of m for ψ over $H_{[c]}$, that is, $\mathcal{Y}(m,\psi) = \mathcal{Z}(m,\psi)$.

Proof. By structural induction over ψ. In the base case, we show the theorem to be true for the MTGL operator *true* to which all MTGCs reduce. We omit the straightforward steps for negation and conjunction.

- *Base case*: *true*.

  By the semantics of MTGL, *true* is always satisfied, hence $\mathcal{Y}(m, \mathit{true}) = \mathcal{Z}(m, \mathit{true}) = \mathbb{R}$.

- *Induction step*: $\psi = \exists(\hat{n},\chi)$.

  Assume that $\mathcal{Y}(\hat{m},\chi) = \mathcal{Z}(\hat{m},\chi)$. We first show that:

  $$\mathcal{Y}(m, \exists(\hat{n},\chi)) \subseteq \bigcup_{\hat{m}\in \underset{\hookrightarrow}{\hat{M}}} \lambda^{\hat{m}} \cap \mathcal{Z}(\hat{m},\chi)$$

  Let $\tau \in \underline{\mathcal{Y}}(m, \exists(\hat{n},\chi))$. By the semantics, a match $\hat{m}$ for $\hat{n}$ exists such that $\hat{m}$ is compatible with the enclosing match m, $\hat{m}$ satisfies χ, and $\max_{\epsilon\in E} \epsilon.\mathit{cts} \leq \tau < \min_{\epsilon\in E} \epsilon.\mathit{dts}$, with E the elements of $\hat{m}$. By definition of $\mathcal{Y}$ and λ, it follows that (i) $\tau \in \underline{\mathcal{Y}}(\hat{m},\chi)$ and by the induction hypothesis $\tau \in \underline{\mathcal{Z}}(\hat{m},\chi)$ (ii) $\tau \in \lambda^{\hat{m}}$, therefore, $\tau \in \lambda^{\hat{m}} \cap \mathcal{Z}(\hat{m},\chi)$ for some $\hat{m}$ compatible with m.

  We proceed with the inclusion $\supseteq$. Let $\tau \in \underline{\mathcal{V}}(\hat{m}, \exists(\hat{n},\chi))$ for some $\hat{m}$ compatible with m. By definition, $\tau \in \lambda^{\hat{m}} \cap \mathcal{Z}(\hat{m},\chi)$. Therefore there exists $\hat{m}$ at τ with $\max_{\epsilon\in E} \epsilon.\mathit{cts} \leq \tau < \min_{\epsilon\in E} \epsilon.\mathit{dts}$ and $\hat{m}$ satisfies χ. Ergo, $\tau \in \underline{\mathcal{Y}}(m, \exists(\hat{n},\chi))$.

  By showing inclusion in both directions, we have shown the sets of time points to be equal.

- *Induction step*: $\psi = \chi \mathrm{U}_I \omega$ and $\ell(I) \neq 0$.

  Assume that $\mathcal{Y}(m,\chi) = \mathcal{Z}(m,\chi)$ and $\mathcal{Y}(m,\omega) = \mathcal{Z}(m,\omega)$. We first show:

  $$\mathcal{Y}(m, \chi \mathrm{U}_I \omega) \subseteq \bigcup_{i\in\mathcal{Z}(m,\omega),\, j\in J_i} j \cap \big((j^{+} \cap i) \ominus I\big)$$

Let $\tau \in \underline{\mathcal{Y}}(m, \chi \mathrm{U}_I \omega)$. By the semantics, there exists a τ' such that ω is satisfied and at least for all $\tau'' \in [\tau, \tau')$ χ is satisfied. Therefore, τ' is in some $i' \in \mathcal{Y}(m, \omega)$ and, by the induction hypothesis, $i' \in \mathcal{Z}(m, \omega)$. Moreover, τ' satisfies the timing constraint I of *until*, thus $\tau' - \tau \in I$. It follows that there exists a $t \in I$ such that $\tau' - \tau = t$, ergo, $\tau = \tau' - t$. Therefore, we obtain that $\tau \in \tau' \ominus I$ and, in turn, $\tau \in i' \ominus I$.

Observe that $[\tau, \tau')$ can't be empty: from $\tau \in \tau' \ominus I$, it follows that $\tau' \in \tau \oplus I$ and, since $\ell(I) \neq 0$ (and by $I \subseteq \mathbb{R}_0^+$, $0 \notin I$), $\tau' > \tau$. Therefore, there exists some $j' \in \mathcal{Y}(m, \chi)$ and, by the induction hypothesis, also in $\mathcal{Z}(m, \chi)$, with $[\tau, \tau') \subseteq j'$. The interval j' is either overlapping or left-adjacent to the i' which contains τ', because $[\tau, \tau'] \subseteq j'^+$ and, from $\tau \in \tau' \ominus I$, we obtain $\tau \in j'^+ \ominus I$. Finally, from $\tau \in j'$ and $\tau \in j'^+ \ominus I$, we obtain, $\tau \in j' \cap (j'^+ \ominus I)$. Since $\tau \in i' \ominus I$, we obtain $\tau \in j' \cap (j'^+ \ominus I) \cap (i' \ominus I)$ and, by $j'^+ \cap i' \neq \emptyset$ and Lemma A.1.1, $\tau \in j' \cap ((j'^+ \cap i') \ominus I)$. It follows that τ is also a member of $\bigcup_{i \in \mathcal{Z}(m,\omega), j \in J_i} j \cap \big((j^+ \cap i) \ominus I\big)$ with J_i the set of all left-adjacent or overlapping intervals for some $i \in \mathcal{Z}(m, \omega)$.

We proceed with showing inclusion $\supseteq$. Let $\tau \in j' \cap ((j'^+ \cap i') \ominus I)$ for one possible pairing of $i' \in \mathcal{Z}(m, \omega)$ and $j' \in J_i$ with J_i defined as previously. That is, $\tau \in j$ and $\tau \in (j'^+ \cap i') \ominus I$. It follows that there exists a $\tau' \in j'^+ \cap i'$ with $\tau' \in \tau \oplus I$, i.e., $\tau' - \tau \in I$. Since $\ell(I) \neq 0$ and $0 \notin I$, $\tau' > \tau$. From $\tau' \in i'$ and $\tau' - \tau \in I$, we deduce that there is a τ' at which ω is satisfied with $\tau' - \tau \in I$. Moreover, from $\tau' \in j'^+$, $\tau \in j'$, and $\tau' > \tau$, we deduce that there is a non-empty $[\tau, \tau') \subseteq j'$ during which χ is satisfied. Therefore, $\chi \mathrm{U}_I \omega$ is satisfied at τ and, by definition of the satisfaction span, $\tau \in \underline{\mathcal{Y}}(m, \chi \mathrm{U}_I \omega)$.

By showing inclusion in both directions, we have shown the sets of time points to be equal.

- *Induction step*: $\psi = \chi \mathrm{U}_I \omega$ and $\ell(I) = 0$. Assume that the induction hypothesis from the previous step holds. We first show:

$$\mathcal{Y}(m, \chi \mathrm{U}_I \omega) \subseteq \bigcup_{i \in \mathcal{Z}(m,\omega)} i \cup \bigcup_{j \in J_i} j \cap \big((j^+ \cap i) \ominus I\big)$$

Let $\tau \in \underline{\mathcal{Y}}(m, \chi \mathrm{U}_I \omega)$. By definition of the satisfaction span, $\chi \mathrm{U}_I \omega$ is satisfied at τ, which, by the semantics, means that there is a time point τ' with $\tau' - \tau \in I$, where ω is satisfied. Observe that by including $0 \in I$, thereby allowing $\tau' - \tau = 0$, the semantics imply a disjunction. If $\tau' - \tau > 0$, it can be shown as before that $\tau \in j' \cap ((j'^+ \cap i') \ominus I)$ for some pairing of $i' \in \mathcal{Y}(m, \omega)$ and $j' \in J_i$ with J_i defined as previously.

If $\tau' - \tau = 0$, it follows that $[\tau, \tau')$ is empty and, since $\tau = \tau'$, $\tau \in i'$ with i' defined as previously. Accounting for both cases of the disjunction, we obtain $\tau \in i' \cup (j' \cap ((j'^+ \cap i') \ominus I))$. It follows that τ is also a member of $\bigcup_{i \in \mathcal{Z}(m,\omega)} i \cup (\bigcup_{j \in J_i} j' \cap ((j'^+ \cap i') \ominus I))$.

We proceed with inclusion $\supseteq$. Let $\tau \in i' \cup (j' \cap ((j'^+ \cap i') \ominus I))$ for some pairing of $i' \in \mathcal{Z}(m, \omega)$ and $j' \in J_i$ with J_i defined as previously. It follows that $\tau \in i' \vee \tau \in j' \cap ((j'^+ \cap i') \ominus I)$. For the left disjunct, since $0 \in I$, there exists τ' such that $\tau' - \tau \in I$, i.e., $\tau' = \tau$ and where $[\tau, \tau')$ is empty. Therefore, τ satisfies *until*. For the right disjunct, we can deduce $\tau \in \mathcal{Y}(m, \chi \mathrm{U}_I \omega)$ as before. We have shown that in each case τ is included in the satisfaction span of until.

By showing inclusion in both directions, we have shown the sets of time points to be equal.

- *Induction step*: $\psi = \chi \mathrm{S}_I \omega$ and $\ell(I) \neq 0$. The step proceeds analogously to *until* and relies on the same induction hypothesis. We first show:

  $$\mathcal{Y}(m, \chi \mathrm{S}_I \omega) \subseteq \bigcup_{i \in \mathcal{Z}(m,\omega),\, j \in J_i} j \cap \big(({}^{+}j \cap i) \oplus I \big)$$

  Let $\tau \in \underline{\mathcal{Y}}(m, \chi \mathrm{S}_I \omega)$. By the semantics, there exists a τ' such that ω is satisfied. Therefore, τ' is in some $i' \in \mathcal{Y}(m, \omega)$ and, by the induction hypothesis, in $\mathcal{Z}(m, \omega)$ too. Moreover, τ' satisfies the timing constraint I of *since*, thus $\tau - \tau' \in I$. It follows that there exists a $t \in I$ such that $\tau - \tau' = t$, thus, $\tau = \tau' + t$. Therefore, we obtain that $\tau \in \tau' \oplus I$ and, in turn, $\tau \in i' \oplus I$.

  Additionally, by the semantics, there exists an interval $(\tau', \tau]$ for which χ is satisfied. Observe that $(\tau', \tau]$ can't be empty: From $\tau \in \tau' \oplus I$, it follows that $\tau' \in \tau \ominus I$ and, since $\ell(I) \neq 0$ (and by $I \subseteq \mathbb{R}_0^+$, $0 \notin I$), $\tau > \tau'$. Therefore, there exists some $j' \in \mathcal{Y}(m, \chi)$ and, by the induction hypothesis, $j' \in \mathcal{Z}(m, \chi)$, with $(\tau', \tau] \subseteq j'$. The interval j' is either overlapping or right-adjacent to the i' which contains τ', because $[\tau', \tau] \subseteq {}^{+}j'$ and, from $\tau' \in \tau \ominus I$, we obtain $\tau \in {}^{+}j' \oplus I$. Finally, from $\tau \in j'$ and $\tau \in {}^{+}j' \oplus I$, we obtain, $\tau \in j' \cap ({}^{+}j' \oplus I)$ and since $\tau \in i' \oplus I$, $\tau \in j' \cap (({}^{+}j' \cap i') \oplus I)$. It follows that τ is also a member of $\bigcup_{i \in \mathcal{Z}(m,\omega), j \in J_i} j' \cap (({}^{+}j' \cap i') \oplus I)$ with J_i the set of all right-adjacent or overlapping intervals for some $i \in \mathcal{Z}(m, \omega)$.

  We proceed with the inclusion $\supseteq$. Let $\tau \in j' \cap (({}^{+}j' \cap i') \oplus I)$ for one possible pairing of $i' \in \mathcal{Z}(m, \omega)$ and $j' \in J_i$ with J_i defined as previously. The time point τ is simultaneously in j' and $({}^{+}j' \cap i') \oplus I$. It follows that there exists a $\tau' \in {}^{+}j' \cap i'$ with $\tau' \in \tau \oplus I$, i.e., $\tau - \tau' \in I$. Since $\ell(I) \neq 0$ and $0 \notin I$, $\tau > \tau'$. From $\tau' \in i'$ and $\tau - \tau' \in I$, we deduce that at τ there is a τ' within the timing constraint I at which ω is satisfied. Moreover, from $\tau' \in {}^{+}j'$, $\tau \in j'$, and $\tau > \tau'$, we deduce that there is a non-empty $(\tau', \tau] \subseteq j'$ during which χ is satisfied. Therefore, $\chi \mathrm{S}_I \omega$ is satisfied at τ and, by definition of the satisfaction span, $\tau \in \underline{\mathcal{Y}}(m, \chi \mathrm{S}_I \omega)$.

  By showing inclusion in both directions, we have shown the sets of time points to be equal.

- *Induction step*: $\psi = \chi \mathrm{S}_I \omega$ and $\ell(I_{\mathrm{U}}) = 0$. Equality can be shown analogously to the corresponding step for *until*.

From the base case and induction steps, it follows that Theorem 3.2.1 holds. □

A.3.2 *Theorem 3.3.1: definite relations imply satisfaction relation over history*

Following is the proof for Theorem 3.3.1, that is, given an MTGC ψ over a pattern n, a sequence of RTM$^{\mathrm{H}}$ instances $h^{H}_{\tau_{\mathcal{D}}}$ with $\mathcal{D} \in \mathbb{N}^+$ the last index, for all $i \in [1, \mathcal{D}] \cap \mathbb{N}^+$, if m a match for n in $H_{[\tau_i]}$ and $\tau \in [0, \tau_i]$, then for all $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$, (i) if $(H_{[\tau_i]}, m, \tau) \vDash^{d} \psi$, then $(H_{[\tau_k]}, m, \tau) \vDash \psi$, and (ii) if $(H_{[\tau_i]}, m, \tau) \vDash^{d}_{F} \psi$, then $(H_{[\tau_k]}, m, \tau) \nvDash \psi$.

Proof. By definition of the RTM$^{\mathrm{H}}$, a match m in $H_{[\tau_i]}$ will be structurally present in all $H_{[\tau_k]}$ with $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$—what may change (once) in future versions of $H_{[\tau_i]}$ is the lifespan of m, i.e., if the *dts* of all matched elements is ∞ and one of these elements is updated to a value less than ∞; even then, this change will not affect the lifespan of m in the period $[0, \tau_i]$, that is, in $H_{[\tau_i]}$, the observation on whether m is present in $\lambda^m \cap [0, \tau_i]$ will never be refuted.

The proof proceeds by mutual structural induction over ψ. In the base case, we show the theorem to be true for the MTGL operator *true*. We omit the straightforward step for conjunction.

- *Base case*: *true*.

  We begin with the definite satisfaction. We assume $(H_{[\tau_i]}, m, \tau) \models^d \mathit{true}$ and show that $(H_{[\tau_k]}, m, \tau) \models \mathit{true}$ for an arbitrary $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$. By the semantics of MTGL, *true* is always satisfied. Therefore, m in $H_{[\tau_k]}$ also satisfies *true* at τ. We have shown that the implication is true.

  We proceed with the definite falsification. Based on the semantics of the definite falsification relation, a match m never falsifies *true*. Therefore, the antecedent $(H_{[\tau_i]}, m, \tau) \models^d_F \mathit{true}$ is false, making the consequent $(H_{[\tau_k]}, m, \tau) \not\models \mathit{true}$ true.

- *Induction step*: $\psi = \neg\chi$.

  We begin with the definite satisfaction. Assume that $(H_{[\tau_i]}, m, \tau) \models^d_F \chi \Rightarrow (H_{[\tau_k]}, m, \tau) \not\models \chi$ for an arbitrary $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$. By the semantics of negation and the definite relations, $(H_{[\tau_i]}, m, \tau) \models^d_F \chi \Leftrightarrow (H_{[\tau_i]}, m, \tau) \models^d \neg\chi$. Similarly, $(H_{[\tau_k]}, m, \tau) \not\models \chi \Leftrightarrow (H_{[\tau_k]}, m, \tau) \models \neg\chi$. Therefore, it also holds that $(H_{[\tau_i]}, m, \tau) \models^d \neg\chi \Rightarrow (H_{[\tau_k]}, m, \tau) \models \neg\chi$.

  We proceed with the definite falsification. Assume that $(H_{[\tau_i]}, m, \tau) \models^d \chi \Rightarrow (H_{[\tau_k]}, m, \tau) \models \chi$. Analogously to the definite satisfaction, $(H_{[\tau_i]}, m, \tau) \models^d \chi \Leftrightarrow (H_{[\tau_i]}, m, \tau) \models^d_F \neg\chi$ and $(H_{[\tau_k]}, m, \tau) \models \chi \Leftrightarrow (H_{[\tau_k]}, m, \tau) \not\models \neg\chi$. Therefore, $(H_{[\tau_i]}, m, \tau) \models^d_F \neg\chi \Rightarrow (H_{[\tau_k]}, m, \tau) \not\models \neg\chi$.

- *Induction step*: $\psi = \exists(\hat{n}, \chi)$.

  Let the induction hypothesis be $(H_{[\tau_i]}, \hat{m}, \tau) \models^d \chi \Rightarrow (H_{[\tau_k]}, \hat{m}, \tau) \models \chi$ and $(H_{[\tau_i]}, \hat{m}, \tau) \models^d_F \chi \Rightarrow (H_{[\tau_k]}, \hat{m}, \tau) \not\models \chi$, where $\hat{m}$ is a match for the pattern $\hat{n}$ and k an arbitrary index in $[i, \mathcal{D}] \cap \mathbb{N}^+$.

  We begin with the definite satisfaction. We assume $(H_{[\tau_i]}, m, \tau) \models^d \exists(\hat{n}, \chi)$ and show this implies $(H_{[\tau_k]}, m, \tau) \models \exists(\hat{n}, \chi)$. Since $(H_{[\tau_i]}, m, \tau) \models^d \exists(\hat{n}, \chi)$, there exists matches m and $\hat{m}$ such that $\hat{m}$ is compatible with m and $\tau \in \lambda^m \cap \lambda^{\hat{m}}$. The matches $m, \hat{m}$ will be structurally present and $\hat{m}$ will be compatible with m in all future versions of $H_{[\tau_i]}$. Moreover, there will be no changes in $\lambda^m, \lambda^{\hat{m}}$ for the period $[0, \tau]$. Also, by the induction hypothesis, $\hat{m}$ satisfies χ at τ. Therefore, by the semantics of the satisfaction relation for *exists*, $(H_{[\tau_k]}, m, \tau) \models \exists(\hat{n}, \chi)$. We have shown that the implication is true.

  We proceed with the definite falsification. We assume that $(H_{[\tau_i]}, m, \tau) \models^d_F \exists(\hat{n}, \chi)$ and show that this implies $(H_{[\tau_k]}, m, \tau) \not\models \exists(\hat{n}, \chi)$. Since $(H_{[\tau_i]}, m, \tau) \models^d_F \exists(\hat{n}, \chi)$, (i) either there exists no $\hat{m}$ in $H_{[\tau_i]}$ such that $\hat{m}$ is compatible with m, or (ii) there exists $\hat{m}$ compatible with m, but $\tau \notin \lambda^m \cap \lambda^{\hat{m}}$, or (iii) there exists $\hat{m}$ compatible with m with $\tau \in \lambda^m \cap \lambda^{\hat{m}}$ but $\hat{m}$ definitely falsifies χ at τ. If (i) is true, it will be true in all future versions of $H_{[\tau_i]}$, as matches cannot be found retrospectively. If (ii) is true, the lifespan of $\lambda^{\hat{m}}$ in the period $[0,\tau_i]$ will not change in all future versions of $H_{[\tau_i]}$. Finally, if (iii) is true, we know from the induction hypothesis that $(\hat{m},\tau) \nvDash \chi$ also over $H_{[\tau_k]}$. Therefore, in any case, $(H_{[\tau_k]},m,\tau) \nvDash \exists(\hat{n},\chi)$. We have shown that the implication is true.

- *Induction step*: $\psi = \chi \mathrm{U}_I \omega$.

  We begin with the definite satisfaction. Induction hypothesis: $(H_{[\tau_i]},m,\tau) \vDash^d \chi \Rightarrow (H_{[\tau_k]},m,\tau) \vDash \chi$ and $(H_{[\tau_i]},m,\tau) \vDash^d \omega \Rightarrow (H_{[\tau_k]},m,\tau) \vDash \omega$ with k an arbitrary index in $[i,\mathfrak{D}] \cap \mathbb{N}^+$.

  We assume $(H_{[\tau_i]},m,\tau) \vDash^d \chi \mathrm{U}_I \omega$ and show this implies $(H_{[\tau_k]},m,\tau) \vDash \chi \mathrm{U}_I \omega$. Since $(H_{[\tau_i]},m,\tau) \vDash^d \chi \mathrm{U}_I \omega$, there exists τ such that $\tau' - \tau \in I$ and $(H_{[\tau_i]},m,\tau') \vDash^d \omega$, and for all $\tau'' \in [\tau,\tau')$ $(H_{[\tau_i]},m,\tau'') \vDash^d \chi$. The decisions for the time point τ' and for all time points τ'' either concern a match or not: if they do concern a match, then they are confined to $[0,\tau_i]$ and remain unaltered throughout the history; if they do not concern a match, e.g., they concern *true* or $\neg$*true*, then they again remain unaltered. Therefore, also over $H_{[\tau_k]}$ it will hold that at τ' $(H_{[\tau_k]},m,\tau') \vDash \omega$, and for every τ'' $(H_{[\tau_k]},m,\tau'') \vDash \chi$. Thus, by the semantics of the satisfaction relation for *until*, $(H_{[\tau_k]},m,\tau) \vDash \chi \mathrm{U}_I \omega$. We have shown that the implication is true.

  We proceed with the definite falsification. Let the induction hypothesis be $(H_{[\tau_i]},m,\tau) \vDash^d_F \chi \Rightarrow (H_{[\tau_k]},m,\tau) \nvDash \chi$ and $(H_{[\tau_i]},m,\tau) \vDash^d_F \omega \Rightarrow (H_{[\tau_k]},m,\tau) \nvDash \omega$.

  We assume $(H_{[\tau_i]},m,\tau) \vDash^d_F \chi \mathrm{U}_I \omega$ and show that this implies $(H_{[\tau_k]},m,\tau) \nvDash \chi \mathrm{U}_I \omega$. Since $(H_{[\tau_i]},m,\tau) \vDash^d_F \chi \mathrm{U}_I \omega$, for all τ' such that $\tau' - \tau \in I$, either (i) $(H_{[\tau_i]},m,\tau') \vDash^d_F \omega$ or (ii) there exists $\tau'' \in [\tau,\tau')$ such that $(H_{[\tau_i]},m,\tau'') \vDash^d \chi$. Regardless of which is the case, i.e., (i) or (ii) or both, analogously to the definite satisfaction, if the decisions for all τ' and at τ'' concern a match, they will remain unaltered, and so will they if they do not concern a match. Therefore, the case will also hold over $H_{[\tau_k]}$. Therefore, $(H_{[\tau_k]},m,\tau) \nvDash \chi \mathrm{U}_I \omega$. We have shown that the implication is true.

- *Induction step*: $\psi = \chi \mathrm{S}_I \omega$.

  The proof proceeds analogously to *until*. We begin with the definite satisfaction. Let the induction hypothesis be $(H_{[\tau_i]},m,\tau) \vDash^d \chi \Rightarrow (H_{[\tau_k]},m,\tau) \vDash \chi$ and $(H_{[\tau_i]},m,\tau) \vDash^d \omega \Rightarrow (H_{[\tau_k]},m,\tau) \vDash \omega$ with k an arbitrary index in $[i,\mathfrak{D}] \cap \mathbb{N}^+$.

  We assume $(H_{[\tau_i]},m,\tau) \vDash^d \chi \mathrm{S}_I \omega$ and show this implies $(H_{[\tau_k]},m,\tau) \vDash \chi \mathrm{S}_I \omega$. Since $(H_{[\tau_i]},m,\tau) \vDash^d \chi \mathrm{S}_I \omega$, there exists τ' such that $\tau - \tau' \in I$ and $(H_{[\tau_i]},m,\tau') \vDash^d \omega$, and for all $\tau'' \in (\tau',\tau]$ $(H_{[\tau_i]},m,\tau'') \vDash^d \chi$. The decisions for the time point τ' and all time points τ'' either concern a match or not: if they do concern a match, then they are confined to $[0,\tau_i]$ and remain unaltered throughout the history; if they do not concern a match, then they will again remain unaltered. Therefore, also over $H_{[\tau_k]}$ it will hold that at τ'

$(H_{[\tau_k]}, m, \tau') \vDash \omega$, and for all τ'' $(H_{[\tau_k]}, m, \tau'') \vDash \chi$. Thus by the semantics of the satisfaction relation for *since*, $(H_{[\tau_k]}, m, \tau) \vDash \chi \mathrm{S}_I \omega$. We have shown that the implication is true.

We proceed with the definite falsification. Let the induction hypothesis be $(H_{[\tau_i]}, m, \tau) \vDash_F^d \chi \Rightarrow (H_{[\tau_k]}, m, \tau) \nvDash \chi$ and $(H_{[\tau_i]}, m, \tau) \vDash_F^d \omega \Rightarrow (H_{[\tau_k]}, m, \tau) \nvDash \omega$.

We assume $(H_{[\tau_i]}, m, \tau) \vDash_F^d \chi \mathrm{S}_I \omega$ and show that this implies $(H_{[\tau_k]}, m, \tau) \nvDash \chi \mathrm{S}_I \omega$. Since $(H_{[\tau_i]}, m, \tau) \vDash_F^d \chi \mathrm{S}_I \omega$, for all τ' such that $\tau - \tau' \in I$, either (i) $(H_{[\tau_i]}, m, \tau') \vDash_F^d \omega$ or (ii) there exists $\tau'' \in (\tau', \tau]$ such that $(H_{[\tau_i]}, m, \tau'') \vDash^d \chi$. Regardless of which is the case, i.e., (i) or (ii) or both, analogously to the definite satisfaction, if the decisions for all τ' and at τ'' concern a match, they will remain unaltered, and so will they if they do not concern a match. Therefore, the case will also hold over $H_{[\tau_k]}$. Therefore, $(H_{[\tau_k]}, m, \tau) \nvDash \chi \mathrm{S}_I \omega$. We have shown that the implication is true.

From the base case and induction steps, it follows that Theorem 3.3.1 holds. □

A.3.3 *Theorem 3.3.2: definite relations are equivalent to satisfaction relation over certain period of history*

Following is the proof for Theorem 3.3.2, that is, given an MTGC ψ over a pattern n, the non-definiteness w window of ψ, and a sequence of RTM$^{\mathrm{H}}$ instances $h^H_{\tau_{\mathcal{D}}}$ with $\mathcal{D} \in \mathbb{N}^+$ the last index, for all $i \in [1, \mathcal{D}] \cap \mathbb{N}^+$, if m a match for n in $H_{[\tau_i]}$ and $\tau \in [0, \tau_i - w]$, then for all $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$, (i) $(H_{[\tau_i]}, m, \tau) \vDash^d \psi$ if and only if $(H_{[\tau_k]}, m, \tau) \vDash \psi$, and (ii) $(H_{[\tau_i]}, m, \tau) \vDash_F^d \psi$ if and only if $(H_{[\tau_k]}, m, \tau) \nvDash \psi$.

By definition of the RTM$^{\mathrm{H}}$, a match m in $H_{[\tau_i]}$ will be structurally present in all $H_{[\tau_k]}$ with $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$—what may change (once) in future versions of $H_{[\tau_i]}$ is the lifespan of m, i.e., if the *dts* of all matched elements is ∞ and one of these elements is updated to a value less than ∞; even then, this change will not affect the lifespan of m in the period $[0, \tau_i]$, that is, in $H_{[\tau_i]}$, the observation on whether m is present in $\lambda^m \cap [0, \tau_i]$ will never be refuted.

Proof. The direction $\Rightarrow$ of the equivalence has been shown by the more general Theorem 3.3.1, which concerned an arbitrary τ. We therefore focus on direction $\Leftarrow$ of the equivalence. As m is present in $H_{[\tau_i]}$, its lifespan λ^m in the period $[0, \tau_i]$ will remain unchanged in subsequent versions of $H_{[\tau_i]}$. In the following, the non-definiteness window w is computed according to Definition 3.3.2.

The proof proceeds by mutual structural induction over ψ. In the base case, we show the theorem to be true for the MTGL operator *true*. We omit the straightforward step for conjunction.

- *Base case*: *true*.

  We begin with the satisfaction. We assume $(H_{[\tau_k]}, m, \tau) \vDash$ *true* for an arbitrary $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$ and $\tau \in [0, \tau_i - w]$ with $w^{nd} = 0$, and show that this implies $(H_{[\tau_i]}, m, \tau) \vDash^d$ *true*. As *true* is always satisfied, m in $H_{[\tau_i]}$ definitely satisfies *true* at τ. Hence, the implication to be true.

We proceed with the falsification. Based on the semantics of satisfaction, a match m never satisfies $\nvDash$ *true*. Therefore, the antecedent $(H_{[\tau_k]}, m, \tau) \nvDash$ *true* is false, making the consequent $(H_{[\tau_i]}, m, \tau) \vDash^d_F$ *true* true.

- *Induction step*: $\psi = \neg\chi$.

  We begin with the satisfaction. Let $(H_{[\tau_k]}, m, \tau) \nvDash \chi \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash^d_F \chi$ for an arbitrary $k \in [i, \mathcal{D}] \cap \mathbb{N}^+$ and $\tau \in [0, \tau_i - w]$ with $w(\neg\chi) = w(\chi)$. By the semantics of negation and the satisfaction relation, $(H_{[\tau_k]}, m, \tau) \nvDash \chi \Leftrightarrow (H_{[\tau_k]} m, \tau) \vDash \neg\chi$. Similarly, $(H_{[\tau_i]}, m, \tau) \vDash^d_F \chi \Leftrightarrow (H_{[\tau_i]}, m, \tau) \vDash^d \neg\chi$. Therefore, it also holds that $(H_{[\tau_k]}, m, \tau) \vDash \neg\chi \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash^d \neg\chi$.

  We proceed with the falsification. Assume $(H_{[\tau_k]}, m, \tau) \vDash \chi \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash^d \chi$. Analogously to the satisfaction, $(H_{[\tau_k]}, m, \tau) \vDash \chi \Leftrightarrow (H_{[\tau_i]}, m, \tau) \nvDash \neg\chi$ and $(H_{[\tau_k]}, m, \tau) \vDash^d \chi \Leftrightarrow (H_{[\tau_k]}, m, \tau) \vDash^d_F \neg\chi$. Therefore, $(H_{[\tau_k]}, m, \tau) \nvDash \neg\chi \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash^d_F \neg\chi$.

- *Induction step*: $\psi = \exists(\hat{n}, \chi)$.

  Let the induction hypothesis be $(H_{[\tau_k]}, \hat{m}, \tau) \vDash \chi \Rightarrow (H_{[\tau_i]}, \hat{m}, \tau) \vDash^d \chi$ and $(H_{[\tau_k]}, \hat{m}, \tau) \nvDash \chi \Rightarrow (H_{[\tau_i]}, \hat{m}, \tau) \vDash^d_F \chi$, where $\hat{m}$ is a match for the pattern $\hat{n}$, k an arbitrary index in $[i, \mathcal{D}] \cap \mathbb{N}^+$, and $\tau \in [0, \tau_i - w]$. The non-definiteness window w is given by $w(\exists(\hat{n}, \chi)) = w(\chi)$.

  We begin with the satisfaction. We assume that $(H_{[\tau_k]}, m, \tau) \vDash \exists(\hat{n}, \chi)$ and show that this implies $(H_{[\tau_i]}, m, \tau) \vDash^d \exists(\hat{n}, \chi)$. Since $(H_{[\tau_k]}, m, \tau) \vDash \exists(\hat{n}, \chi)$, there exists matches m and $\hat{m}$ in $H_{[\tau_k]}$ such that $\hat{m}$ is compatible with m and $\tau \in \lambda^m \cap \lambda^{\hat{m}}$. The match m is present in $H_{[\tau_i]}$ and, according to the induction hypothesis, the match $\hat{m}$ is also present to m in $H_{[\tau_i]}$. As the matches are structurally the same, $\hat{m}$ is also compatible with m in $H_{[\tau_i]}$. Moreover, as there are no changes in $\lambda^m, \lambda^{\hat{m}}$ for the period $[0, \tau_i]$, $\tau \in \lambda^m \cap \lambda^{\hat{m}}$ over $H_{[\tau_i]}$. We also know that $\tau \le \tau_i$ and, by the induction hypothesis, that $\hat{m}$ satisfies χ at τ. Therefore, by the semantics of the definite satisfaction relation for *exists*, $(H_{[\tau_i]}, m, \tau) \vDash^d \exists(\hat{n}, \chi)$. We have shown that the implication is true.

  We proceed with the falsification. We assume that $(H_{[\tau_k]}, m, \tau) \nvDash \exists(\hat{n}, \chi)$ and show that this implies $(H_{[\tau_i]}, m, \tau) \vDash^d_F \exists(\hat{n}, \chi)$. Since $(H_{[\tau_k]}, m, \tau) \nvDash \exists(\hat{n}, \chi)$, (i) either there exists no $\hat{m}$ in $H_{[\tau_k]}$ such that $\hat{m}$ is compatible with m, or (ii) there exists $\hat{m}$ compatible with m, but $\tau \notin \lambda^m \cap \lambda^{\hat{m}}$, or (iii) there exists $\hat{m}$ compatible with m with $\tau \in \lambda^m \cap \lambda^{\hat{m}}$ but $\hat{m}$ falsifies χ at τ. If (i) is true, it will be true in all future versions of $H_{[\tau_i]}$, as matches cannot be found retrospectively. If (ii) is true, the lifespan of $\lambda^{\hat{m}}$ in the period $[0, \tau_i]$ will not change in all future versions of $H_{[\tau_i]}$. Finally, if (iii) is true, we know from the induction hypothesis that $(\hat{m}, \tau) \vDash^d_F \chi$ also over $H_{[\tau_i]}$ and that $\tau \le \tau_i$. Therefore, in any case, $(H_{[\tau_i]}, m, \tau) \vDash^d_F \exists(\hat{n}, \chi)$. We have shown that the implication is true.

- *Induction step*: $\psi = \chi \mathrm{U}_I \omega$.

  We begin with the satisfaction. Let the induction hypothesis be $(H_{[\tau_k]}, m, \tau) \vDash \chi \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash^d \chi$ and $(H_{[\tau_k]}, m, \tau) \vDash \omega \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash^d \omega$ with k

an arbitrary index in $[i, \mathcal{D}] \cap \mathbb{N}^+$ and $\tau \in [0, \tau_i - w]$. The non-definiteness window w is given by $max(w(\chi), w(\omega)) + r(I)$.

We assume $(H_{[\tau_k]}, m, \tau) \models \chi \mathrm{U}_I \omega$ and show $(H_{[\tau_i]}, m, \tau) \models^d \chi \mathrm{U}_I \omega$. Since $(H_{[\tau_k]}, m, \tau) \models \chi \mathrm{U}_I \omega$, there exists τ' such that $\tau' - \tau \in I$ and $(H_{[\tau_k]}, m, \tau') \models \omega$, and for all $\tau'' \in [\tau, \tau')$ $(H_{[\tau_k]}, m, \tau'') \models \chi$. From $\tau \in [0, \tau_i - w]$ and $\tau' \in [\tau + \ell(I), \tau + r(I)]$, it follows that $\tau' \leq \tau_i - max(w(\chi), w(\omega))$. Based on this and the induction hypothesis, $(H_{[\tau_i]}, m, \tau') \models^d \omega$. Moreover, as τ' stems from a period outside the non-definiteness window of ω, the decision at τ', whether it concerns a match or not, will remain unaltered once made.

The decision at τ' as well as the preceding period $[\tau, \tau')$ are also outside the non-definiteness window of χ. Thus, all $\tau'' \in [\tau, \tau')$ stem from a period covered by $H_{[\tau_i]}$, and decisions for χ made in this period are definite. Therefore, for all $[\tau + \ell(I), \tau + \tau')$ $(H_{[\tau_i]}, m, \tau'') \models^d \chi$, and, by the definite semantics, $(H_{[\tau_i]}, m, \tau) \models^d \chi \mathrm{U}_I \omega$. We have shown that the implication is true.

We proceed with the falsification. Let the induction hypothesis be that $(H_{[\tau_k]}, m, \tau) \not\models \chi \Rightarrow (H_{[\tau_i]}, m, \tau) \models^d_F \chi$ and $(H_{[\tau_k]}, m, \tau) \not\models \omega \Rightarrow (H_{[\tau_i]}, m, \tau) \models^d_F \omega$.

We assume $(H_{[\tau_k]}, m, \tau) \not\models \chi \mathrm{U}_I \omega$ and show $(H_{[\tau_i]}, m, \tau) \models^d_F \chi \mathrm{U}_I \omega$. Since $(H_{[\tau_k]}, m, \tau) \not\models \chi \mathrm{U}_I \omega$, it holds that for all τ' such that $\tau' - \tau \in I$ either (i) $(H_{[\tau_k]}, m, \tau') \not\models \omega$ or (ii) there exists $\tau'' \in [\tau, \tau')$ such that $(H_{[\tau_k]}, m, \tau'') \models \chi$. Regardless of which is the case, i.e., (i) or (ii) or both, analogously to the satisfaction, the decisions for all τ' and at τ'' stem from a period that is covered by $H_{[\tau_i]}$, and decisions made in this period regarding χ and ω are definite. Therefore, the case will also hold over $H_{[\tau_i]}$. Therefore, $(H_{[\tau_i]}, m, \tau) \models^d_F \chi \mathrm{U}_I \omega$. We have shown that the implication is true.

- *Induction step*: $\psi = \chi \mathrm{S}_I \omega$.

  We begin with the satisfaction. Let the induction hypothesis be $(H_{[\tau_k]}, m, \tau) \models \chi \Rightarrow (H_{[\tau_i]}, m, \tau) \models^d \chi$ and $(H_{[\tau_k]}, m, \tau) \models \omega \Rightarrow (H_{[\tau_i]}, m, \tau) \models^d \omega$ with k an arbitrary index in $[i, \mathcal{D}] \cap \mathbb{N}^+$ and $\tau \in [0, \tau_i - w]$. The non-definiteness window w is given by $max(w(\chi), w(\omega))$.

  We assume $(H_{[\tau_k]}, m, \tau) \models \chi \mathrm{S}_I \omega$ and show $(H_{[\tau_i]}, m, \tau) \models^d \chi \mathrm{S}_I \omega$. Since $(H_{[\tau_k]}, m, \tau) \models \chi \mathrm{S}_I \omega$, there exists τ' such that $\tau - \tau' \in I$ and $(H_{[\tau_k]}, m, \tau') \models \omega$, and for all $\tau'' \in (\tau', \tau]$ $(H_{[\tau_k]}, m, \tau'') \models \chi$. From $\tau \in [0, \tau_i - w]$ and $\tau' \in [\tau - r(I), \tau - \ell(I)]$, it follows that $\tau' \leq \tau_i - max(w(\chi), w(\omega))$. Hence, the decision at τ' can already be made over $H_{[\tau_i]}$, and, moreover, as τ' stems from a period outside the non-definiteness window of ω, the decision at τ', whether it concerns a match or not, will remain unaltered once made. Therefore, $(H_{[\tau_i]}, m, \tau') \models^d \omega$. The decision at τ' as well as the succeeding period $(\tau', \tau]$ is also outside the non-definiteness window of χ. Thus, all $\tau'' \in (\tau', \tau]$ stem from a period covered by $H_{[\tau_i]}$, and decisions for χ made in this period are definite. Therefore, for all $\tau'' \in (\tau', \tau]$ $(H_{[\tau_i]}, m, \tau'') \models^d \chi$, and, by the definite semantics, $(H_{[\tau_i]}, m, \tau) \models^d \chi \mathrm{S}_I \omega$. We have shown that the implication is true.

We proceed with the falsification. Let the induction hypothesis be that $(H_{[\tau_k]}, m, \tau) \nvDash \chi \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash_F^d \chi$ and $(H_{[\tau_k]}, m, \tau) \nvDash \omega \Rightarrow (H_{[\tau_i]}, m, \tau) \vDash_F^d \omega$.

We assume $(H_{[\tau_k]}, m, \tau) \nvDash \chi \mathrm{S}_I \omega$ and show $(H_{[\tau_i]}, m, \tau) \vDash_F^d \chi \mathrm{S}_I \omega$. Since $(H_{[\tau_k]}, m, \tau) \nvDash \chi \mathrm{S}_I \omega$, it holds that for all τ' such that $\tau - \tau' \in I$ either (i) $(H_{[\tau_k]}, m, \tau') \nvDash \omega$ or (ii) there exists $\tau'' \in (\tau', \tau]$ such that $(H_{[\tau_k]}, m, \tau'') \vDash \chi$. Regardless of which is the case, i.e., (i) or (ii) or both, analogously to the satisfaction, the decisions for all τ' and at τ'' stem from a period that is covered by $H_{[\tau_i]}$, and decisions made in this period regarding χ and ω are definite. Therefore, the case will also hold over $H_{[\tau_i]}$. Therefore, $(H_{[\tau_i]}, m, \tau) \vDash_F^d \chi \mathrm{S}_I \omega$. We have shown that the implication is true.

From the base case and induction steps, it follows that Theorem 3.3.2 holds. □

A.3.4 *Theorem 3.3.3: equality of definite spans and definite computations for satisfaction and falsification*

Following is the proof for Theorem 3.3.3, i.e., given a match m over an RTM$^{\mathrm{H}}$ $H_{[\tau]}$ and an MTGC ψ, the definite satisfaction span $\mathcal{Y}^d$ of m for ψ over $H_{[\tau]}$ is given by the definite satisfaction computation $\mathcal{Z}^d$ of m for ψ over $H_{[\tau]}$ in Definition 3.3.3, that is, $\mathcal{Y}^d(m, \psi) = \mathcal{Z}^d(m, \psi)$. Moreover, the definite falsification span $\mathcal{F}$ of m for ψ over $H_{[\tau]}$ is given by the definite falsification computation F of m for ψ over $H_{[\tau]}$ in Definition 3.3.3, that is, $\mathcal{F}(m, \psi) = F(m, \psi)$.

Proof. The proof for the definite satisfaction span $\mathcal{Z}^d$ proceeds almost identically to the proof in Theorem 3.2.1 for $\mathcal{Z}$, i.e., by structural induction over ψ, and therefore omitted. For *true*, conjunction, *exists*, *until*, and *since* in Definition 3.3.3, inclusion can be shown in both directions—the proof for the negation relies on a reasoning analogous to the one presented below for negation for the definite falsification span.

The proof for the definite falsification F is based on the application of $F = \mathbb{R} \setminus (\mathcal{Z}^d \uplus X)$ for each MTGL operator—which follows from $\mathbb{R} = \mathcal{Y}^d \uplus \mathcal{F} \uplus X$. The unknown span X for *true* is $X = \emptyset$, whereas for *exists*, by definition of the RTM$^{\mathrm{H}}$ $H_{[\tau]}$, it is $X = (\tau, \infty)$. If F is known, it can be used to compute $\mathcal{Z}^d \uplus X$.

- $\psi = true$: From Equation 3.9 in Definition 3.3.3, we have $\mathcal{Z}^d(m, true) = \mathbb{R}$, therefore $F(m, true) = \emptyset$.

- $\psi = \neg\chi$: It holds that

  $$\overline{F}(m, \neg\chi) = \mathcal{Z}^d(m, \neg\chi) \uplus X(m, \neg\chi)$$

  and

  $$\overline{\mathcal{Z}^d}(m, \chi) = \mathcal{Z}^d(m, \neg\chi) \uplus X(m, \neg\chi)$$

  Therefore,

  $$F(m, \neg\chi) = \overline{\overline{\mathcal{Z}^d}}(m, \chi) = \mathcal{Z}^d(m, \chi)$$

- $\psi = \chi \wedge \omega$: Let each time point that does not definitely falsify the MTGC a that χ encloses to be assumed to satisfy the a. In practice, this includes all time points in $\mathcal{Z}^d(m,\chi) \uplus X(m,\chi)$ for a. Subtracting this maximal satisfaction span from the time domain $\mathbb{R}$ yields the set of time points that definitely falsify χ. Let the satisfaction span of ω be defined analogously. If the satisfaction span of conjunction is computed based on these maximal satisfaction spans of χ and ω, i.e., by $(\mathcal{Z}^d(m,\chi) \uplus X(m,\chi)) \cap (\mathcal{Z}^d(m,\omega) \uplus X(m,\omega))$, the definite falsification span of conjunction can be computed analogously.

$$\begin{aligned} F(m,\chi \wedge \omega) &= \mathbb{R} \setminus \big((\mathcal{Z}^d(m,\chi) \uplus X(m,\chi)) \cap (\mathcal{Z}^d(m,\omega) \uplus X(m,\omega))\big) \\ &= \mathbb{R} \setminus \big((\mathbb{R} \setminus F(m,\chi)) \cap (\mathbb{R} \setminus F(m,\omega))\big) \\ &= F(m,\chi) \cup F(m,\omega) \end{aligned}$$

- $\psi = \exists(\hat{n},\chi)$: Let τ be the time point of the RTM$^{\mathrm{H}}$ $H_{[\tau]}$. As $\mathcal{Z}(m,\exists(\hat{n},\chi))$ is known and $X(m,\exists(\hat{n},\chi)) = (\tau,\infty)$, to obtain the falsification computation, we can directly solve $\mathbb{R} \setminus (\mathcal{Z}^d \uplus X)$.

$$\begin{aligned} F(m,\exists(\hat{n},\chi)) &= \mathbb{R} \setminus \big(\mathcal{Z}^d(m,\exists(\hat{n},\chi)) \cup (\tau,\infty)\big) \\ &= \big(\mathbb{R} \setminus (\tau,\infty)\big) \cap \big(\mathbb{R} \setminus \mathcal{Z}^d(m,\exists(\hat{n},\chi))\big) \\ &= (-\infty,\tau] \cap \big(\mathbb{R} \setminus \mathcal{Z}^d(m,\exists(\hat{n},\chi))\big) \end{aligned}$$

- $\psi = \chi \mathrm{U}_I \omega$ and $0 \notin I$: The computation for *until* relies on the reasoning explained in the case of conjunction. The satisfaction span of *until* is computed based on the maximal satisfaction spans of ω, i.e., $\mathcal{Z}^d(m,\omega) \uplus X(m,\omega)$, and χ, that is, J_i^X is obtained by $\mathcal{Z}^d(m,\omega) \uplus X(m,\omega)$ and $\mathcal{Z}^d(m,\chi) \uplus X(m,\chi)$, thus the *until* satisfaction span is similarly maximal. Therefore, complementing this maximal satisfaction span yields all time points that definitely falsify *until*. Therefore, we have:

$$F(m,\chi \mathrm{U}_I \omega) = \mathbb{R} \setminus \left(\bigcup_{i \in \mathcal{Z}^d(m,\omega) \cup X(m,\omega),\, j \in J_i^X} j \cap \big((j^+ \cap i) \ominus I\big) \right)$$

- $\psi = \chi \mathrm{U}_I \omega$ and $0 \in I$: The reasoning is similar to the case where $0 \notin I$.

- $\psi = \chi \mathrm{S}_I \omega$ and $0 \notin I$: The case proceeds analogously to the corresponding case of *until*.

- $\psi = \chi \mathrm{S}_I \omega$ and $0 \in I$: The case proceeds analogously to the corresponding case of *until*.

By showing that $\mathcal{Y}^d(m,\psi) = \mathcal{Z}^d(m,\psi)$ and the equations for $F(m,\psi)$, we have shown that theorem holds. □

A.3.5 *Lemma 4.1.1: Equality of satisfaction span set and TGDN result*

Following is the proof for Lemma 4.1.1, which states that, given an RTM$^{\mathrm{H}}$ $H_{[\tau]}$, a context pattern n, an MTGC ψ over n, and a TGDN g characterized by the output of $\mathcal{C}(n,\psi)$, the satisfaction span set Σ_ψ in $H_{[\tau]}$ is equal to the result $\mathcal{G}_\psi$ of g, that is:

$$\Sigma_\psi(H_{[\tau]}) = \mathcal{G}_\psi(H_{[\tau]})$$

Proof. We show the lemma by structural induction over ψ and the TGDNs that operation $\mathcal{C}(n, \psi)$ constructs. For each ψ, we obtain g based on $\mathcal{C}$ and show that the result yielded by the TGDN g is equal to Σ_ψ. The proof refers to items from Section 4.1.3. In the base case, we show the theorem to be true for the MTGL operator *true* to which all MTGCs reduce. We omit the straightforward step for *since*.

- *Base case*: *true*.

  We first show inclusion $\subseteq$. Let t be a tuple $(m, \mathcal{Z}(m, true))$ in Σ_{true} for the context pattern n and the MTGC *true* with m a match for n. In this case, $\mathcal{Z}(m, true) = \mathbb{R}$. The TGDN g obtained by $\mathcal{C}(n, true)$ consists of a single BMR r whose LHS is the same as n. Thus, r is applied at least for m and a marking node a for m is created. According to item (vi), the duration of the marking node $a.d$ is $\mathbb{R}$, thus equal to $\mathcal{Z}(m, true)$. The TGDN result $\mathcal{G}_{true}$ contains the result of the operation obt for all marking nodes, thus also for a which obtains m. This m is paired with $a.d$. Therefore, there is a tuple t' in $\mathcal{G}$ such that $t' = t$. We have shown inclusion $\subseteq$.

  We proceed with inclusion $\supseteq$. As previously mentioned, the constructed g consists of a single rule r whose LHS is n. Let $(obt(a), a.d)$ be a tuple in $\mathcal{G}$ with a the marking node created by r for a match m for n, and $a.d = \mathbb{R}$ the duration of the marking node. By definition, the satisfaction span set Σ contains all matches for n paired with their satisfaction span, therefore also m. The satisfaction span of this match is $\mathbb{R}$ (see Equation 3.2), i.e., it is equal to $a.d$. We have shown inclusion $\supseteq$.

- *Induction step*: $\psi = \neg\chi$.

  We first show inclusion $\subseteq$. Assume that $\Sigma_\chi = \mathcal{G}_\chi$. Let $t \in \Sigma_\psi$ such that $t = (m, \mathcal{Z}(m, \neg\chi))$. The same match m must be also included in Σ_χ, i.e., there exists a tuple $t' = (m, \mathcal{Z}(m, \chi))$. Let g_χ be the TGDN constructed by $\mathcal{C}(n, \chi)$. By the induction hypothesis, there exists a tuple $t'' = (obt(a), a.d)$ in $\mathcal{G}_\chi$ such that $t'' = t'$, i.e., $obt(a) = m$ and $a.d = \mathcal{Z}(m, \chi)$. Let g be the TGDN constructed by $\mathcal{C}(n, \neg\chi)$, which consists of a sub-tree corresponding to g plus the rules added by the step for negation in Section 4.1.3, that is: a multi-rule μ whose LHS is the pattern $\underset{\sim}{n}_\chi$, a kernel rule k whose LHS is n, and the dependencies (μ, r_χ) and (k, μ), where r_χ is the root of g. The existence of t'' implies that r_χ is applied at least once for m and creates a marking node. Therefore, at least one copy of μ is created for $\underset{\sim}{m}_\chi$. This copy is glued in the kernel rule k. The LHS of the rule k is n therefore the rule is also applied for m and factors in its computation of the duration $a.d$ of the marking node v the copy of μ for m. The duration $a.d$ is given by $\mathbb{R} \setminus a.d$. As $a.d$ is equal to $\mathcal{Z}(m, \chi)$, this computation is effectively equal to $\mathbb{R} \setminus \mathcal{Z}(m, \chi)$, which, according to Theorem 3.2.1, is equal to $\mathcal{Z}(m, \neg\chi)$. Therefore, there exists a tuple t''' in $\mathcal{G}_\psi$ such that $t''' = t$. We have shown inclusion $\subseteq$.

  We now proceed with inclusion $\supseteq$. Let g be the TGDN defined above. Let $(obt(a), a.d)$ be a tuple u in $\mathcal{G}_\psi$ with a a marking node created by the root k of g. The LHS of k is n, therefore $obt(a)$ yields a match m for n. As shown before, the duration $a.d$ is equal to $\mathcal{Z}(m, \neg\chi)$. The satisfaction span set Σ_ψ includes all matches for n paired with their satisfaction span $\mathcal{Z}$. Thus, Σ_ψ

also includes a match m. Therefore, there exists a tuple $u' \in \Sigma_\psi$ with $u' = u$. We have shown inclusion $\subseteq$.

- *Induction step*: $\psi = \chi \wedge \omega$.

  We first show inclusion $\subseteq$. Assume that $\Sigma_\chi = \mathcal{G}_\chi$ and $\Sigma_\omega = \mathcal{G}_\omega$. Let t be a tuple $(m, \mathcal{Z}(m, \psi))$ in $\Sigma_{\chi \wedge \omega}$ with m a match for the context pattern n. The same match m must be also included in Σ_χ and Σ_ω, as χ and ω are also MTGCs over n. Therefore, there exists a tuple $t'_\chi = (m, \mathcal{Z}(m, \chi)) \in \Sigma_\chi$ and a tuple $t'_\omega = (m, \mathcal{Z}(m, \omega)) \in \Sigma_\omega$. Let g_χ be the TGDN constructed by $\mathcal{C}(n, \chi)$ and g_ω be the TGDN constructed by $\mathcal{C}(n, \omega)$. By the induction hypothesis, there exists a tuple $t''_\chi = (obt(a_\chi), a_\chi.d)$ in $\mathcal{G}_\chi$ such that $t''_\chi = t'_\chi$, i.e., $obt(a_\chi) = m$ and $a_\chi.d = \mathcal{Z}(m, \chi)$. Similarly, there exists a tuple $t''_\omega = (obt(a_\omega), a_\omega.d)$ in $\mathcal{G}_\omega$ such that $t''_\omega = t'_\omega$. Let g be the TGDN constructed by $\mathcal{C}(n, \chi \text{U}_I \omega)$, which consists of the sub-trees corresponding to g_χ and g_ω plus the rule added by item (iii), that is: a BMR r whose LHS is $\underset{\sim}{n}_{\chi,\omega}$ and whose dependencies are both r_χ, i.e., the root of g_χ, and r_ω, i.e., the root of g_ω. The existence of t''_χ and t''_ω imply that both of these rules were applied and created a marking node. Hence, since a match m for n exists and so do the marking nodes for r_χ and r_ω, r is applied at least once for m and creates a marking node a. The duration of the marking node is computed by Equation 3.4, based on the duration a_χ and a_ω. According to Theorem 3.2.1, these are equal to $\mathcal{Z}(m, \chi)$ and $\mathcal{Z}(m, \omega)$, respectively; thus, the computation by r is equal to $\mathcal{Z}(m, \chi \wedge \omega)$. Therefore, there exists a tuple t''' in $\mathcal{G}_\psi$ such that $t''' = t$. We have shown inclusion $\subseteq$.

  We now proceed with inclusion $\supseteq$. Let g be the TGDN defined above. Let $(obt(a), a.d)$ be a tuple u in $\mathcal{G}_\psi$ with a a marking node created by the root r of g. The LHS of r is $\underset{\sim}{n}_{\chi,\omega}$, therefore $obt(a)$ yields a match m for n. As shown before, the duration $a.d$ is equal to $\mathcal{Z}(a, \chi \wedge \omega)$. The satisfaction span set Σ_ψ includes all matches for n paired with their satisfaction span $\mathcal{Z}$. Thus, Σ_ψ also includes m. Therefore, there exists a tuple $u' \in \Sigma_\psi$ with $u' = u$. We have shown inclusion $\subseteq$.

- *Induction step*: $\psi = \exists(\hat{n}, \chi)$.

  We first show inclusion $\subseteq$. Assume that, given the input $(\hat{n}, \chi)$, $\mathcal{C}(\hat{n}, \chi)$ produces a TGDN $\hat{g}$ such that the satisfaction span set Σ_χ for χ over $\hat{n}$ is given by the result $\mathcal{G}_\chi$ of $\hat{g}$, i.e., $\Sigma_\chi = \mathcal{G}_\chi$. Let t be a tuple $(m, \mathcal{Z}(m, \exists(\hat{n}, \chi)))$ in Σ_ψ with m a match for n. Then t' is a tuple $(\hat{m}, \mathcal{Z}(\hat{m}, \chi))$ in Σ_χ with $\hat{m}$ compatible with m. By the induction hypothesis, there also exists a tuple $t'' = (obt(\hat{a}), \hat{a}.d)$ in the result $\mathcal{G}_\chi$ of $\hat{g}$ with $\hat{a}$ the marking node created for a match $\hat{m}$ such that $t'' = t'$, that is, $\hat{a}.d = \mathcal{Z}(\hat{m}, \chi)$. This implies that the root r_χ of $\hat{g}$ creates a marking node for $\hat{m}$, where the LHS of r_χ is $\hat{n}$ extended with the marking node created by the dependency of r_χ.

  Consider the TGDN g obtained by the operation $\mathcal{C}(n, \exists(\hat{n}, \chi))$. Before the item in the operation $\mathcal{C}_{rec}$ for *exists* is performed (see Definition 4.1.2), the dependency r_χ is obtained by performing $\mathcal{C}_{rec}(\hat{n}, \chi)$. Notice that this operation effectively constructs $\hat{g}$ and returns the root of $\hat{g}$, i.e., r_χ. Then, the item in $\mathcal{C}$ for *exists* is performed, and three rules are added: i) a BMR $r_{\hat{n}}$ (ii) a multi-rule μ (iii) a kernel rule k; k is dependent on μ, μ is dependent on $r_{\hat{n}}$, and $r_{\hat{n}}$ is dependent on r_χ. The rule k is the root of g. The LHS of $r_{\hat{n}}$ is the extended pattern $\underset{\sim}{\hat{n}}_{r_\chi}$, i.e., the pattern $\hat{n}$ extended by the marking node created by r_χ. As mentioned previously, $\mathcal{G}_\chi$ of $\hat{g}$ contains a tuple $t'' = t'$, therefore the rule r_χ created a marking node for a match for the pattern $\hat{n}$ extended by the dependency of r_χ. It follows that $r_{\hat{n}}$ also finds a match (both a match $\hat{m}$ for $\hat{n}$ and a marking node for r_χ exists) and creates a marking node whose duration is the intersection of all matched nodes, i.e., the duration is equal to the temporal validity $\lambda^{\hat{m}} \cap \mathcal{Z}(\hat{m}, \chi)$ of $\hat{m}$—see item (i).

After $r_{\hat{n}}$ has been applied, the dependency of μ is satisfied, so μ is applied. The LHS of μ searches for marking nodes created by $r_{\hat{n}}$. At least one such marking node exists (the one created for $\hat{m}$ in the previous step) and therefore at least one copy of μ is created. This copy is glued in the kernel rule k. The multi-rule μ creates no marking nodes. The LHS of k is n. Since the tuple t in Σ_ψ contains a match m for n, k is also applied and finds m. Moreover, this match is compatible with $\hat{m}$—as the elements in $\hat{m}$ is a subset of the elements in m. The duration of the marking node a created for m is computed as the union of the duration of all marking nodes matched by μ, i.e.,:

$$\bigcup_{\hat{m} \in \hat{M}} \lambda^{\hat{m}} \cap \mathcal{Z}(\hat{m}, \chi)$$

with $\hat{M}$ the set of all matches for $\hat{n}$ compatible with m. According to Theorem 3.2.1, this union is equal to $\mathcal{Z}(m, \exists(\hat{n}, \chi))$. Therefore, the tuple $t''' = (obt(a), a.d))$ is included in the result $\mathcal{G}_\psi$ obtained by the root of g, i.e., k. As $obt(a) = m$ and $a.d = \mathcal{Z}(m, \exists(\hat{n}, \chi))$, $t''' = t$. We have shown inclusion $\subseteq$.

We now proceed with inclusion $\supseteq$. Let g be the TGDN defined above. Let $(obt(a), a.d)$ be a tuple u in $\mathcal{G}_\psi$ with a a marking node created by the root k of g. The LHS of k is identical to n, therefore $obt(a)$ yields a match m for n. As shown before, the duration $a.d$ is equal to $\mathcal{Z}(m, \exists(\hat{n}, \chi))$. The satisfaction span set Σ_ψ includes all matches for n paired with their satisfaction span $\mathcal{Z}$. Thus, Σ_ψ also includes the match m. Therefore, there exists a tuple $u' \in \Sigma_\psi$ with $u' = u$. We have shown inclusion $\subseteq$.

- *Induction step*: $\psi = \chi \mathrm{U}_I \omega$.

  We first show inclusion $\subseteq$. Assume that $\Sigma_\chi = \mathcal{G}_\chi$ and $\Sigma_\omega = \mathcal{G}_\omega$. Let $t \in \Sigma_\psi$ such that $t = (m, \mathcal{Z}(m, \chi \mathrm{U}_I \omega))$ with m a match for n. The same match m must be also included in Σ_χ and Σ_ω, as χ and ω are also MTGCs over n. Therefore, there exists a tuple $t'_\chi = (m, \mathcal{Z}(m, \chi)) \in \Sigma_\chi$ and a tuple $t'_\omega = (m, \mathcal{Z}(m, \omega)) \in \Sigma_\omega$. Let g_χ be the TGDN constructed by $\mathcal{C}(n, \chi)$ and g_ω be the TGDN constructed by $\mathcal{C}(n, \omega)$. By the induction hypothesis, there exists a tuple $t''_\chi = (obt(a_\chi), a_\chi.d)$ in $\mathcal{G}_\chi$ such that $t''_\chi = t'_\chi$, i.e., $obt(a_\chi) = m$ and $a_\chi.d = \mathcal{Z}(m, \chi)$. Similarly, there exists a tuple $t''_\omega = (obt(a_\omega), a_\omega.d)$ in $\mathcal{G}_\omega$ such that $t''_\omega = t'_\omega$.

  Let g be the TGDN constructed by $\mathcal{C}(n, \chi \mathrm{U}_I \omega)$, which consists of the subtrees corresponding to g_χ and g_ω plus the rules added by item (iv), that is: a multi-rule μ whose LHS is $\underset{\sim}{n}_\chi$, a kernel rule k whose LHS is $\underset{\sim}{n}_\omega$; μ is dependent on r_χ, i.e., the root of g_χ, and k dependent on μ. The LHS of the kernel rule k includes the marking node created by the root of g_ω. The existence of t''_χ implies that r_χ is applied at least for the match m and creates a marking node. Therefore, at least one copy of μ is created for $\underset{\sim}{m}_\chi$. This copy is glued in the kernel rule k. The existence of t''_ω implies that r_ω is applied at least for m and therefore creates a marking node. The LHS of the rule k is $\underset{\sim}{n}_\omega$, and since a match m for n exists and so does a marking node for r_ω, k is also applied for $\underset{\sim}{m}_\omega$. The duration of the marking node created by k is based on the RHS of Equation 3.6, where the duration $a_\chi.d$ and $a_\omega.d$ are used. According to Theorem 3.2.1, these are equal to $\mathcal{Z}(m,\chi)$ and $\mathcal{Z}(m,\omega)$, respectively; thus, the computation by k is equal to $\mathcal{Z}(m,\chi \mathrm{U}_I \omega)$. Therefore, there exists a tuple t''' in $\mathcal{G}_\psi$ such that $t''' = t$. We have shown inclusion $\subseteq$.

We now proceed with inclusion $\supseteq$. Let g be the TGDN defined above.b Let $(obt(a), a.d)$ be a tuple u in $\mathcal{G}_\psi$ with a a marking node created by the root k of g. The LHS of k is $\underset{\sim}{n}_\omega$, therefore $obt(a)$ yields a match m for n. As shown before, the duration $a.d$ is equal to $\mathcal{Z}(a,\chi \mathrm{U}_I \omega)$. The satisfaction span set Σ_ψ includes all matches for n paired with their satisfaction span $\mathcal{Z}$. Thus, Σ_ψ also includes m. Therefore, there exists a tuple $u' \in \Sigma_\psi$ with $u' = u$. We have shown inclusion $\subseteq$.

From the base case and induction steps, it follows that Lemma 4.1.1 holds. □

A.3.6 *Lemma 4.1.2: equality of definite satisfaction and falsification span set and TGDN definite result*

First, we elaborate on the proof for Lemma 4.1.2, which states that, given an RTM$^\mathrm{H}$ $H_{[\tau]}$, a context pattern n, an MTGC ψ over n, and a TGDN g^d characterized by the output of $\mathcal{C}^d(n,\psi)$, the definite satisfaction and falsification span set Σ^d_ψ in $H_{[\tau]}$ is given by the definite result $\mathcal{G}^d_\psi$ of g^d, that is:

$$\Sigma^d_\psi(H_{[\tau]}) = \mathcal{G}^d_\psi(H_{[\tau]})$$

Proof. The proof proceeds almost identically to the proof for Lemma 4.1.1. The similarity is justified by rule definitions in Section 4.1.3 and the proof for Lemma 4.1.1, where it becomes apparent that rule applications in a TGDN mainly ensure that matches and their dependencies are structurally present; the duration computation does not affect the network execution.

This proof relies on the created marking nodes storing both the definite satisfaction computation $\mathcal{Z}^d$ and definite falsification computation F: For example, for the computation of $\mathcal{Z}^d$ for negation, the created marking node uses the F of the dependency, and vice versa; for the computation of the definite falsification for *until* and *since*, the created marking node computes $\mathcal{Z}^d \uplus X$ of the dependencies based on their F. □

A.3.7 *Theorem 4.1.2: equality of TVGDN definite result and query definite answer set*

We proceed with the proof for Theorem 4.1.2, i.e., given an RTM$^\mathrm{H}$ $H_{[\tau]}$, a query (n,ψ) in $\mathcal{L}_\mathrm{T}$, and a TVGDN $g^{d,\mathcal{V}}$ for (n,ψ), the result $\mathcal{G}^{d,\mathcal{V}}$ of $g^{d,\mathcal{V}}$ over $H_{[\tau]}$ is equal to the definite query answer set $\mathcal{T}^d$ of (n,ψ) over $H_{[\tau]}$.

Proof. The proof proceeds almost identically to Theorem 4.1.1. The argumentation is only extended to refer to both $\mathcal{Z}^d$ and F, and their corresponding duration attributes $a.d$ and $a.d^F$—where at least one of them must be not empty. □

A.3.8 *Theorem 4.3.1: equality of aggregation of projected answer sets over a sequence of pruned RTM^H instances and a sequence of complete RTM^H instances*

Following is the proof for Theorem 4.3.1, which states that, given a query $\zeta := (n, \psi)$ in $\mathcal{L}_{\mathrm{T}}$, a history $h_{[\tau_{\mathcal{D}}]}$ with $\mathcal{D} \in \mathbb{N}^+$, and the sequences $h^H_{\tau_{\mathcal{D}}}$, $h^P_{\tau_{\mathcal{D}}}$ of unpruned and pruned RTM$^{\mathrm{H}}$ instances corresponding to $h_{[\tau_{\mathcal{D}}]}$, the aggregation of the projected answer set $\mathcal{T}^\pi$ over $h^H_{\tau_{\mathcal{D}}}$ is equal to the aggregation of $\mathcal{T}^\pi$ over $h^P_{\tau_{\mathcal{D}}}$, that is:

$$\bigcup_{i=1}^{\mathcal{D}} \mathcal{T}^\pi_{\tau_i}(H_{[\tau_i]}) = \bigcup_{i=1}^{\mathcal{D}} \mathcal{T}^\pi_{\tau_i}(P_{[\tau_i]})$$

Proof. The proof proceeds by induction over $\mathcal{D}$. In the following, we omit the time point of $\mathcal{T}^\pi$ as it is clear from the context. Moreover, for a tuple $u := (m, \mathcal{V})$ we refer to the match of the tuple by m_u and the temporal validity of the tuple by $\mathcal{V}_u$.

- *Base case:* $\mathcal{D} = 1$.

  Holds by definition of $H_{[\tau_1]}$ and $P_{[\tau_1]}$.

- *Induction step:* $\mathcal{D} > 1$.

  We first need to show that:

  $$\bigcup_{i=1}^{j} \mathcal{T}^\pi(H_{[\tau_j]}) = \bigcup_{i=1}^{j} \mathcal{T}^\pi(P_{[\tau_j]}) \implies \bigcup_{i=1}^{j+1} \mathcal{T}^\pi(H_{[\tau_{j+1}]}) = \bigcup_{i=1}^{j+1} \mathcal{T}^\pi(P_{[\tau_{j+1}]})$$

  We first show inclusion $\subseteq$, that is:

  $$\bigcup_{i=1}^{j} \mathcal{T}^\pi(H_{[\tau_j]}) \subseteq \bigcup_{i=1}^{j} \mathcal{T}^\pi(P_{[\tau_j]}) \implies \bigcup_{i=1}^{j+1} \mathcal{T}^\pi(H_{[\tau_{j+1}]}) \subseteq \bigcup_{i=1}^{j+1} \mathcal{T}^\pi(P_{[\tau_{j+1}]})$$

  The proof proceeds by tracking the match of a tuple and its temporal validity in the projected answer set of a non-pruned RTM$^{\mathrm{H}}$, via three steps, in (1) the regular answer set over the non-pruned RTM$^{\mathrm{H}}$, and from there to (2) the regular answer set of a pruned RTM$^{\mathrm{H}}$, and from there to (3) the projected answer set of the pruned RTM$^{\mathrm{H}}$.

  (1) Let t be a tuple $(m, \mathcal{V})$ such that

  $$(m, \mathcal{V}) \in \bigcup_{i=1}^{j+1} \mathcal{T}^\pi(H_{[\tau_{j+1}]})$$

  for the query (n, ψ). It either holds that

  $$t \in \bigcup_{i=1}^{j} \mathcal{T}^\pi(H_{[\tau_j]})$$

or not. In the former case, it must also hold that

$$t \in \bigcup_{i=1}^{j} \mathcal{T}^{\pi}(P_{[\tau_j]})$$

(from the antecedent), and therefore,

$$t \in \bigcup_{i=1}^{j+1} \mathcal{T}^{\pi}(P_{[\tau_{j+1}]})$$

In the latter case, t is inserted by the latest answer set, therefore $t \in \mathcal{T}^{\pi}(H_{[\tau_{j+1}]})$. By definition of $\mathcal{T}^{\pi}$, it follows that there exists a $t' \in \mathcal{T}(H_{[\tau_{j+1}]})$ with $m_{t'}$ the same as m_t and $\mathcal{V}_{t'} \cap [\tau_j - \mathcal{W}^r, \tau_{j+1}] = \mathcal{V}_t$ with $\mathcal{W}^r$ the involvement window of ψ from the query. Step 1 is done.

(2) Henceforth, we denote the interval $[\tau_j - \mathcal{W}, \tau_{j+1}]$ by Γ. By definition of $\mathcal{W}^r$, an element e may affect a temporal validity at an arbitrary time point x only if $x \in e.\lambda \ominus \mathcal{W}^r$ with λ the lifespan of e, i.e., if $e.dts \geq x - \mathcal{W}^r$. For $\mathcal{V}_{t'}$, the earliest time point for which this may happen is $\ell(\Gamma)$. Hence, if we replace x with $\ell(\Gamma)$, we get that all elements e with $e.dts \geq \tau_j - 2\mathcal{W}^r$ or, since the current time point is τ_{j+1}, all e with $e.dts \in [\tau_j - 2\mathcal{W}^r, \infty]$ may affect $\mathcal{V}_{t'}$. By definition of a pruned RTM$^{\mathrm{H}}$, all such elements are contained in $P_{[\tau_{j+1}]}$. Therefore, m_t exists in $P_{[\tau_{j+1}]}$ and elements in m that could affect the temporal validity in $\mathcal{V}_{t'} \cap \Gamma$ are unchanged. From this, it follows that there is a $t'' \in \mathcal{T}(P_{[\tau_{j+1}]})$ with $m_{t''}$ the same as m_t and $\mathcal{V}_{t''} = \mathcal{V}_{t'}$. Step 2 is done.

(3) By definition of $\mathcal{T}^{\pi}$, it follows that there is a tuple t''' in $\mathcal{T}^{\pi}(P_{[\tau_{j+1}]})$ with $m_{t'''}$ the same as m_t and $\mathcal{V}_{t'''}$ equal to $\mathcal{V}_{t''} \cap \Gamma$ which is equal to $\mathcal{V}_t$. Ergo, $t''' = t$, and also

$$t \in \bigcup_{i=1}^{j+1} \mathcal{T}^{\pi}(P_{[\tau_{j+1}]})$$

Step 3 is done and we have shown the inclusion $\subseteq$.

We proceed with showing the inclusion $\supseteq$, that is:

$$\bigcup_{i=1}^{j} \mathcal{T}^{\pi}(P_{[\tau_j]}) \subseteq \bigcup_{i=1}^{j} \mathcal{T}^{\pi}(H_{[\tau_j]}) \implies \bigcup_{i=1}^{j+1} \mathcal{T}^{\pi}(P_{[\tau_{j+1}]}) \subseteq \bigcup_{i=1}^{j+1} \mathcal{T}^{\pi}(H_{[\tau_{j+1}]})$$

Let t be a tuple $(m, \mathcal{V})$ such that:

$$(m, \mathcal{V}) \in \bigcup_{i=1}^{j+1} \mathcal{T}^{\pi}(P_{[\tau_{j+1}]})$$

for the query (n, ψ). If it was true that

$$t \in \bigcup_{i=1}^{j} \mathcal{T}^{\pi}(P_{[\tau_j]})$$

it can be shown as in the other direction that

$$t \in \bigcup_{i=1}^{j+1} \mathcal{T}^{\pi}(H_{[\tau_{j+1}]})$$

Otherwise, $t \in \mathcal{T}^{\pi}(P_{[\tau_{j+1}]})$. We follow the steps from the previous direction in reverse.

(1) By definition of $\mathcal{T}^{\pi}$ it follows that there is a t' in $\mathcal{T}(P_{[\tau_{j+1}]})$ with $m_{t'}$ the same as m_t and $\mathcal{V}_{t'} = \mathcal{V}_t \cap \Gamma$.

(2) By definition of pruning, all elements e which were pruned in $P_{[\tau_{j+1}]}$ yet contained in $H_{[\tau_{j+1}]}$ must have a $dts < \tau_i - 2\mathcal{W}^r$, i.e., $e.dts \in [0, \tau_i - 2\mathcal{W}^r)$. These elements can not affect the elements of $m_{t'}$ or its temporal validity $\mathcal{V}_{t'}$. Therefore, $\mathcal{T}(H_{[\tau_{j+1}]})$ contains a tuple t'' with $m_{t''}$ the same as $m_t{}'$ and $\mathcal{V}_{t''} = \mathcal{V}_{t'}$.

(3) Finally, by definition of $\mathcal{T}^{\pi}$, there is a tuple $t''' \in \mathcal{T}^{\pi}(H_{[\tau_{j+1}]})$ with m its match and $\mathcal{V}_{t'''} = \mathcal{V}_{t''}$ therefore equal to $\mathcal{V}_t$. Ergo, $t''' = t$, and therefore, t is also in $\bigcup_{i=1}^{j+1} \mathcal{T}(H_{[\tau_{j+1}]})$.

We have shown inclusion $\supseteq$.

By the base case and the induction step, it follows that the theorem holds. $\square$

B EXPERIMENTAL EVALUATION SUPPLEMENT

This chapter is a supplement to the experimental evaluation presented in Chapter 5. Section B.1 describes the prototypical implementation in further detail. Section B.2 presents the E2P specification and query formulations for the SHS case-study in Section 5.2. Section B.3 presents the E2P specification and query formulations for the SNB experiments in Section 5.3.

B.1 IMPLEMENTATION

Section B.1.1 and Section B.1.2 present the grammars and more information on ITQL and E2P, respectively. Section B.1.3 describes the operation modes of the EMF plugin.

B.1.1 *The InTempo Query Language*

The (slightly simplified) grammar of ITQL is shown in Figure B.1, where the type ID in is a specific type of STRING. InTempo relies on the *Story Pattern Matcher* [66, 114] for pattern matching, an EMF tool from the tool support for SDs in [57]—see also the discussion in Section 6.1.

The main entity in SDs are *Story Nodes*, which are connected by control flow edges. Story Nodes contain *Story Patterns*, which represent structural fragments that should be matched. ITQL supports the definition of Story Patterns both by a direct definition (<*StoryPattern*>) and by a declaration (<*Declaration*>). Declarations aim at making the specification of queries more concise—see Listing B.2 for an example. They are used to building Story Patterns during the loading of the ITQL file. Story Patterns are passed to the Story Pattern Matcher which performs the actual matching.

To facilitate the formulation of intervals, ITQL support shorthand notations for referring to days (“`d`"), months (“`m`"), and years (“`y`"); moreover, the language supports the specification of open-closed (“`oc`”), closed-open (“`co`”), and open-open intervals (“`oo`”).

InTempo offers an Xtext [22] editor for ITQL which supports completion suggestions for element types and validation of the query syntax.

B.1.2 *The Events-to-Patterns Specification Language*

The grammar of E2P is shown in Figure B.2. The main challenge in designing E2P was the capability to translate from an event-specific to a platform-specific reference system. For example, within the log, events might be linked together by referring to the identifiers of other events but once stored in an RTM^H this identifier, or *id* for short, becomes a regular vertex attribute which cannot be used to immediately identify the vertex.

```
⟨Condition⟩   ::= ⟨And⟩ ( ‘declarations’ ‘{’ ⟨Declaration⟩* ‘}’ )?
⟨And⟩         ::= ⟨Until⟩ ( ‘AND’ ⟨Until⟩ )?
⟨Until⟩       ::= ⟨Since⟩ ( ‘U’ interval=⟨Interval⟩ ⟨Since⟩ )?
⟨Since⟩       ::= ⟨BasicCondition⟩ ( ‘S’ interval=⟨Interval⟩ ⟨BasicCondition⟩ )?
⟨BasicCondition⟩ ::= ‘true’ | ⟨Not⟩ | ⟨Exists⟩ | ‘(’ ⟨And⟩ ‘)’ | ⟨Proxy⟩
⟨Not⟩         ::= ‘!’ ⟨BasicCondition⟩
⟨Exists⟩      ::= ‘E’ ( ( ⟨StoryPattern⟩ | ⟨Proxy⟩ ) ⟨Binding⟩?
                  | ‘(’ ( ⟨StoryPattern⟩ | ⟨Proxy⟩ ) ⟨Binding⟩? ( ‘,’ ⟨And⟩ )? ‘)’ )
⟨StoryPattern⟩ ::= ID ‘{’ ( ⟨StoryPatternObject⟩ | ⟨StoryPatternLink⟩
                  | ‘[’ ⟨StringExpression⟩ ‘]’ )* ‘}’
⟨StoryPatternObject⟩ ::= ID ‘:’ ⟨ScopedType⟩
⟨StoryPatternLink⟩ ::= ⟨QualifiedName⟩ ‘-’ ⟨ScopedFeature⟩ ‘->’ ⟨QualifiedName⟩
⟨StringExpression⟩ ::= ID ‘:’ STRING
⟨ScopedType⟩ ::= ID ( ‘::’ ID )?
⟨ScopedFeature⟩ ::= ID
⟨Proxy⟩       ::= ‘$’ ⟨QualifiedName⟩
⟨Binding⟩     ::= ‘[’ ⟨Mapping⟩* ‘]’
⟨Mapping⟩     ::= ID ‘->’ ID
⟨Declaration⟩ ::= ⟨NamedExpression⟩ | ⟨StoryPattern⟩
⟨NamedExpression⟩ ::= ID ‘:’ ⟨And⟩
⟨QualifiedName⟩ ::= ID ( ‘.’ ID )*
⟨Interval⟩    ::= ‘[’ (‘0’..‘9’)* (‘d’ | ‘m’ | ‘y’)? ‘,’ ((‘0’..‘9’)* (‘d’ | ‘m’ | ‘y’)? | ‘*’) ‘]’ (‘oc’ | ‘co’ |
                  ‘oo’)?
```

Figure B.1: The grammar of ITQL

```
⟨Event⟩        ::= STRING ':{' ⟨Action⟩* '}'
⟨Action⟩       ::= ⟨Add⟩ | ⟨AddRef⟩ | ⟨Delete⟩ | ⟨DeleteRef⟩ | ⟨Modify⟩
⟨Add⟩          ::= 'adds' ( ⟨ReferralByName⟩ ':' )? ID
                   ( '»'⟨Commit⟩ )? ( '[' ⟨AttributeAssignment⟩*']' )?
⟨Delete⟩       ::= 'deletes' ⟨ReferralByRetrieval⟩
⟨Modify⟩       ::= 'modifies' ⟨Referral⟩ '[' ⟨AttributeAssignment⟩* ']'
⟨AddRef⟩       ::= 'adds-ref' ⟨Referral⟩ '-' ID '->' ⟨Referral⟩
⟨AddRefs⟩      ::= 'adds-refs' ⟨Referral⟩ '-' ID '->' ⟨BatchReferralByRetrieval⟩
⟨DeleteRef⟩    ::= 'deletes-ref' ID ⟨Referral⟩ '->' ⟨Referral⟩
⟨Commit⟩       ::= ID '(' ⟨Value⟩ ')'
⟨Referral⟩     ::= ⟨ReferralByName⟩ | ⟨ReferralByRetrieval⟩
⟨ReferralByRetrieval⟩ ::= '$' ID '(' ⟨Value⟩ ')'
⟨BatchReferralByRetrieval⟩ ::= '$' ID '(' ⟨Value⟩ '+)'
⟨ReferralByName⟩ ::= ID
⟨AttributeAssignment⟩ ::= ID '=' ⟨Value⟩
⟨Value⟩        ::= ⟨StringValue⟩ | ⟨ParameterValue⟩
⟨StringValue⟩ ::= STRING
⟨ParameterValue⟩ ::= '*p' INT (('~*p' INT)? ('+' INT)? ('-' INT)?);
```

Figure B.2: The grammar of E2P

Hence, E2P implements an *indexing system* so that specifications may designate specific nodes to be *indexed*: either for *event-wide* use, i.e., within an event mapping, for example, when an event adds multiple elements and elements added later need to refer to those added earlier; or *specification-wide*, i.e., outside the event mapping, for example, when an element needs to refer to a vertex that has been added by a previous event. The indexing system is realized by <*Commit*> and <*Referral*>. The token ">>" marks that the node being added should be indexed, i.e., *committed*, for specification-wide reference. The succeeding parameters denote the name of the index as well as the key the vertex should be added under. See Listing B.1 for an example, where the created instance of `PMonitoringService` is added to an index named *PMServices*. The key under which this new instance is indexed is the value of the first attribute of the event. This is achieved by the <*ParameterValue*> statement, which retrieves a value of the parameter in a given position of the event, specified by the "*" token. For conveniently creating unique ids for links that are encoded as vertices, this statement also supports the concatenation of values, using the token "~", and basic arithmetic operations, i.e., addition and subtraction, which are performed upon the retrieval of the value. Declared indices, if any, are created during initialization of InTempo. Event-wide references are marked by the *name* before a type, e.g., the `pm` in `pm:PMonitoringService`.

Referrals to indexed node can be made either by retrieving a node from an index (*ReferralByRetrieval*), using the token "`$`" together with the name of the index and the key, or by referring to a named reference (*ReferralByName*), e.g., the *adds-ref* statement in Listing B.1 where an edge is created from the `pm` to the instance `pr` of type `Probe`. Nodes added during initialization (see `init` in Listing B.1), are indexed by default and can be thus referred to by name by all mappings. *BatchReferralByRetrieval* allows for referring to multiple indexed nodes at the same time, i.e., a functionality required by the SNB files where a newly created message may have many tags, all passed in the same event—see Listing B.8.

As with ITQL, InTempo offers an Xtext editor for E2P which supports completion suggestions for element types in the imported metamodel and validation of the mapping syntax.

B.1.3 *Operation Modes*

InTempo supports two operation modes, assuming different settings: One mode, the RTMHAnalysis, supports the evaluation of a temporal query over an RTM^H, whereas the other, the LogAnalysis, assumes that, instead of being captured by an RTM^H, data about the system execution are captured by events in a log. In practice, the LogAnalysis triggers an RTMHAnalysis on an internally created RTM^H, following the mapping of an event to model changes performed on the RTM^H in question.

InTempo can be used either via the EMF user interface or via an API. If the user interface is used, RTMHAnalysis expects a user-specified ITQL query (in a file with .itql extension) and a user-provided RTM^H, i.e., a persisted instance of an EMF model in the standard XMI format. LogAnalysis expects an ITQL query, a log file containing comma-separated values (see Table B.1), and the E2P mapping (a file with an .e2p extension).

If the API is used, it is expected that the inputs are passed programmatically. In case of RTMHAnalysis, the RTM^H model can be passed either via an XMI file or as an object. In case of LogAnalysis, InTempo can be used to either iteratively analyze a log file or a single event; in the latter case, it is assumed that an external application, i.e., an adaptation engine, is using InTempo and managing the processing of events generated by a system. In the API operation mode, the RTM^H as well as query answers are always available to external tools for further processing.

B.2 SMART HOSPITAL SYSTEM

B.2.1 *Log Synthesis*

The logs discussed in this section are available in an online repository in [133]. The following text is a summary of the documentation accompanying the logs in the repository. We refer the reader to this documentation for more details on the employed statistical methods. The logs presented here have been extended with deletion events—see Section 5.2.1; these logs are available in [134] and were used in the experimental evaluation of the SHS case-study in Section 5.2.

The logs are based on a real log which records 1050 patient cases diagnosed with sepsis and admitted to the emergency ward of a hospital over the course of 1.5 years [108]. The log records *patient trajectories*, i.e., the course of patients that have been diagnosed with sepsis during their stay in the hospital. The trajectory

Table B.1: Exemplary Log Entries

```
1,ER Sepsis Triage,1414013868
2,ER Sepsis Triage,1414022808
1,IV Antibiotics,1414927128
```

is tracked via events recorded by the Enterprise Resource Planning system of the hospital. The logs are used for checking the compliance of trajectories to the medical guideline in [126], which is also used in Section 5.2.

In its original form, the log contains several types of medical and logistical events. We create a *simplified event log* which retains only three relevant types of events: the event `ER Sepsis Triage`, indicating the beginning of a trajectory, i.e., the admission of a patient diagnosed with sepsis; the event `IV Antibiotics`, indicating the intravenous administration of antibiotics to treat sepsis; and the `Release` events which are variants of an event which largely corresponds to the end of the patient treatment. Each trajectory in the real log is represented by a *trace* (a sequence of events) and starts with a recorded `ER Sepsis Triage`. The original log contains one trace that does not start with an `ER Sepsis Triage` event which is omitted from our simplified version. Hence, the simplified version contains 1049 trajectories. `IV Antibiotics` and `Release` events may follow an `ER Sepsis Triage` event.

The original log also records several metadata per event. The simplified log contains only the *trace id* of an event, i.e., an identification number which captures the trace to which the event belongs, and the *timestamp*, i.e., the date and time (with millisecond precision) on which the event occurred. Entries in the simplified log are *comma-separated values* of the following form: `<TRACE ID>,<EVENT>,<TIMESTAMP>`. Timestamp values are transformed into a UNIX timestamp. Table B.1 shows exemplary log entries from the simplified event log; this log is henceforth referred to as *real*.

The *inter-arrival time* (IAT) between two events capture the elapsed time between the occurrence of these two events. We denote the IAT between two `ER Sepsis Triage` events by IAT_T. Similarly, the IAT is calculated for pairs of the events `ER Sepsis Triage` and `IV Antibiotics` (IAT_{SA}) and `ER Sepsis Triage` and `Release` (IAT_{SR}). For each of these calculated measures, a data series was created that consists of 1048 IATs values, i.e., one for each pair of events. Based on statistical methods, a probability distribution was derived that best fitted each data series. Following the procedure described in [133], these distributions enabled the generation of x10 and x100; the logs cover the same period of time as the real log and increase the trajectory density (approx.) 10 and 100 times, respectively, while preserving the statistical characteristics of the real log.

B.2.2 *E2P Specification for the SHS*

The E2P specification used for mapping events into the SHS log files to model-specific changes is shown in Listing B.1. The model-specific changes are based on the SHS metamodel in Figure 2.1, which is imported at the very top of the E2P specification.

```
import 'http://mdelab.de/intempo/examples/shs/1.0'

init:{adds hospital:SHSRoot}

"ER Sepsis Triage":{adds pm:PMonitoringService >> PMServices(*p1) [ID=*p1]
    adds pr:Probe >> SProbes(*p1) [ID=*p1 status="sepsis" cts=*p3]
    adds-ref pm -probes-> pr
    adds-ref hospital -ownedServices-> pm}

"IV Antibiotics":{adds d:DrugService >> DServices(*p1) [ID=*p1]
    adds pr:Probe >> AProbes(*p1) [ID=*p1 status="anti" cts=*p3]
    adds-ref d -probes-> pr
    adds-ref hospital -ownedServices-> d}

"Release A":{adds pr:Probe >> RProbes(*p1) [ID=*p1 status="release" cts=*p3]
    adds-ref $PMServices(*p1) -probes-> pr}

"Release B":{adds pr:Probe >> RProbes(*p1) [ID=*p1 status="release" cts=*p3]
    adds-ref $PMServices(*p1) -probes-> pr}

"Release C":{adds pr:Probe >> RProbes(*p1) [ID=*p1 status="release" cts=*p3]
    adds-ref $PMServices(*p1) -probes-> pr}

"Release D":{adds pr:Probe >> RProbes(*p1) [ID=*p1 status="release" cts=*p3]
    adds-ref $PMServices(*p1) -probes-> pr}

"Release E":{adds pr:Probe >> RProbes(*p1) [ID=*p1 status="release" cts=*p3]
    adds-ref $PMServices(*p1) -probes-> pr}

"Delete Sepsis Probe":{modifies $SProbes(*p1) [dts=*p3]}

"Delete PMonitoringService":{modifies $PMServices(*p1) [dts=*p3]}

"Delete Antibiotics Probe":{modifies $AProbes(*p1) [dts=*p3]}

"Delete Drug Service":{modifies $DServices(*p1) [dts=*p3] }

"Delete Release Probe":{modifies $RProbes(*p1) [dts=*p3]}
```

Listing B.1: The mapping of events to model modifications for the SHS

B.2.3 *Queries in ITQL*

The queries MG1 and MG2 are described in Section 4.4.1. Their specification in ITQL is shown in Listing B.2 and Listing B.3, respectively. The specifications refer to entity types in the SHS metamodel in Figure 2.1.

B.2.4 *Formulas in MFOTL*

MonPoly uses a specification language which is based on the MFOTL. As MTGL, MFOTL is a metric temporal logic, which allows for preserving the main structure of the MTGC ψ_{MG1}. The query MG1 $:= (u_1, \psi_{\text{MG1}})$ is translated into a formula $\exists u_1 \wedge \neg \psi_{\text{MG1}}$ in MFOTL. Lifespans and patterns require a special handling which we described in Section 5.4.1. The formula in MFOTL corresponding to MG1 is showed in Listing B.4. This construction has to be guarded by an (atomic) event, which we also ensure; in practice, this means MonPoly ignores some deletion events compared to InTempo and Hawk. A temporal operator with no metric interval is a shorthand for the same operator with

```
import 'http://mdelab.de/intempo/examples/shs/1.0'

($u1, !(true U[0,3600] E $u11))

declarations{
    u1{pr1:Probe
       pm:PMonitoringService
       s:SHSService
       pm -probes-> pr1
       s -connected-> pm
       [OCL:"pr1.status='sepsis'"]}

    u11{pr2:Probe
        pm:PMonitoringService
        s:SHSService
        d:DrugService
        d -probes-> pr2
        s -connected-> pm
        s -connected-> d
        [OCL:"pr2.status='anti'"]
        [OCL:"pm.pID=d.pID"]}
}
```

Listing B.2: The query MG1 in ITQL

```
import 'http://mdelab.de/intempo/examples/shs/1.0'

($u1, !(!(E $u12) U[0,3600] E $u11))

declarations{
    u1{pr1:Probe
        pm:PMonitoringService
        s:SHSService
        pm -probes-> pr1
        s -services-> pm
        [OCL:"pr1.status='sepsis'"]}

    u11{pr2:Probe
        pm:PMonitoringService
        s:SHSService
        d:DrugService
        d -probes-> pr2
        s -services-> pm
        s -services-> d
        [OCL:"pr2.status='anti'"]
        [OCL:"pm.pID=d.pID"]}

    u12{pr3:Probe
        pm:PMonitoringService
        s:SHSService
        pm -probes-> pr3
        s -services-> pm
        [OCL:"pr3.status='release'"]}
}
```

Listing B.3: The query MG2 in ITQL

```
EXISTS q,x,y,z .
(probe(x,"sepsis") AND ONCE (
  ((NOT del_mprobes(x,y)) SINCE mprobes(x,y))
  AND ONCE (
    ((NOT del_mconnected(y,z)) SINCE mconnected(y,z))
    AND ONCE (
      ((NOT del_mservice(y,q)) SINCE mservice(y,q))
      AND ONCE shs(z)))))
AND NOT EVENTUALLY[0,3600] EXISTS a,b,p .
(probe(a,"anti")
AND ONCE (
  ((NOT del_dprobes(a,b)) SINCE dprobes(a,b))
  AND ONCE (
    ((NOT del_dconnected(b,z)) SINCE dconnected(b,z))
    AND ONCE (
      ((NOT del_dservice(b,p)) SINCE dservice(b,p))
      AND p=q AND ONCE shs(z)))))
```

Listing B.4: The translation of the query MG1 into a formula in MFOTL

the interval $[0, \infty)$. The logs used in the MonPoly experiments track time in seconds which is also the assumed granularity in the timing constraints of the formulas.

The query MG2 is analogously translated into the formula shown in Listing B.5. As mentioned previously, MG2 cannot be monitored by MonPoly.

```
EXISTS q,x,y,z .
(probe(x,"sepsis") AND ONCE (
  ((NOT del_mprobes(x,y)) SINCE mprobes(x,y))
  AND ONCE (
    ((NOT del_mconnected(y,z)) SINCE mconnected(y,z))
    AND ONCE (
      ((NOT del_mservice(y,q)) SINCE mservice(y,q))
      AND ONCE shs(z)))))
AND NOT EXISTS a,b,p,d .
(NOT probe(d,"release") AND ONCE
  ((NOT del_mprobes(x,y)) SINCE mprobes(x,y))
  AND ONCE (
    ((NOT del_mconnected(y,z)) SINCE mconnected(y,z))
    AND ONCE (
      ((NOT del_mservice(y,q)) SINCE mservice(y,q))
      AND ONCE (shs(z))))))
UNTIL[0,3600] EXISTS a,b,p .
(probe(a,"anti")
AND ONCE (
  ((NOT del_dprobes(a,b)) SINCE dprobes(a,b))
  AND ONCE (
    ((NOT del_dconnected(b,z)) SINCE dconnected(b,z))
    AND ONCE (
      ((NOT del_dservice(b,p)) SINCE dservice(b,p))
      AND p=q AND ONCE shs(z)))))
```

Listing B.5: The translation of the query MG2 into a formula in MFOTL. The formula cannot be monitored by MonPoly

B.2.5 *Queries in EOL*

Hawk assigns timestamps to versions based on the timestamp of the corresponding git commit, which is tracked in milliseconds. Therefore, timing constraints in EOL are adjusted accordingly. The queries MG1 and MG2 are translated into EOL as described in Section 5.4.2. To the extent possible, the translations follow the structure of the corresponding MTGCs. The translations for MG1 and MG2 are presented in Listing B.6 and Listing B.7, respectively.

```
return
Probe.getVersionsFrom(Probe.latest.time-7200000)
.all.flatten.asSet
.select(p | p.status="sepsis").select(
 p | p.eContainer.select(
  pmon | pmon.eContainer.select(
   shs | not shs.eventually(
    shsp | shsp.connected.exists(
     d | d.isTypeOf(DrugService) and d.pID=pmon.pID
     and d.probes.exists(
     p2 | p2.status="anti"
     and p2.earliest.time <= p.earliest.time+3600000
     and p2.earliest.time > p.earliest.time))))
 .notEmpty)
.notEmpty)
.collect(p | Map{"pid"=p.eContainer.pID, "time"=p.earliest.time})
.flatten;
```

Listing B.6: The query MG1 in EOL

```
return
Probe.getVersionsFrom(Probe.latest.time-7200000)
.all.flatten.asSet
.select(p | p.status="sepsis").select(
 p | p.eContainer.select(
  pmon | pmon.eContainer.select(
   shs | not shs.eventually(
    shsp | shsp.connected.exists(
     d | d.isTypeOf(DrugService) and d.pID=pmon.pID
     and d.probes.exists(
     p2 | p2.status="anti"
     and p2.earliest.time <= p.earliest.time+3600000
     and p2.earliest.time > p.earliest.time
     and not Probe.getVersionsBetween(p.earliest.time,p2.earliest.time)
     .all.flatten.asSet.exists(pp | pp.eContainer=pmon and pp.status="release"))))
  .notEmpty)
.notEmpty)
.collect(p | Map{"pid"=p.eContainer.pID,"time"=p.earliest.time})
.flatten;
```

Listing B.7: The query MG2 in EOL

B.3 SOCIAL NETWORK BENCHMARK

B.3.1 *E2P Specification for the SNB*

The E2P specification that maps events in the SNB logs into model-specific changes is shown in Listing B.8. The model-specific changes are based on the SNB metamodel in Figure 5.4.

```
import 'http://mdelab.de/intempo/examples/ldbc_snb/1.0'
```

```
init:{adds model:LdbcSNBModel}

//Static
"CREATE_TAGCLASS":{adds tc:TagClass >> TagClasses(*p2) [ID=*p2 name=*p3]
    adds-ref model -ownedTagClasses-> tc}

"CREATE_TAG":{adds t:Tag >> Tags(*p2) [ID=*p2 name=*p3]
    adds-ref t -hasType-> $TagClasses(*p4)
    adds-ref model -ownedTags-> t}

"CREATE_COUNTRY":{adds country:Country >> Countries(*p2) [ID=*p2 name=*p3]
    adds-ref model -ownedCountries-> country}

"CREATE_CITY":{adds city:City >> Cities(*p2) [ID=*p2]
    adds-ref city -isPartOf-> $Countries(*p3)
    adds-ref model -ownedCities-> city}

"CREATE_UNIVERSITY":{adds uni:University >> Unis(*p2) [ID=*p2]
    adds-ref uni -isLocatedIn-> $Cities(*p3)
    adds-ref model -ownedUniversities-> uni}

"CREATE_COMPANY":{adds com:Company [ID=*p2]
    adds-ref com -isLocatedIn-> $Countries(*p3)
    adds-ref model -ownedCompanies-> com}

//Dynamic
"CREATE_PERSON":{adds p:Person >> Persons(*p2) [ID=*p2 firstName=*p3]
    adds-ref p -isLocatedIn-> $Cities(*p4)
    adds-refs p -hasInterest-> $Tags(*p5+)
    adds-ref model -ownedPersons-> p}

"DELETE_PERSON":{modifies $Persons(*p2) [dts=*p1]}

"CREATE_MEMBER":{adds hmL:HasMemberLink >> MemberLinks(*p3~*p2)
    adds-ref hmL -forum-> $Forums(*p2)
    adds-ref hmL -person-> $Persons(*p3)
    adds-ref model -ownedHasMemberLinks-> hmL}

"DELETE_MEMBER":{modifies $MemberLinks(*p2~*p3) [dts=*p1]}

"CREATE_FORUM":{adds f:Forum >> Forums(*p2) [ID=*p2]
    adds-ref model -ownedForums-> f}

"DELETE_FORUM":{modifies $Forums(*p2) [dts=*p1]}

"CREATE_LIKES":{adds lL:LikesLink >> LikesLinks(*p2~*p3)
    adds-ref lL -person-> $Persons(*p2)
    adds-ref lL -likes-> $Messages(*p3)
    adds-ref model -ownedLikesLinks-> lL}

"DELETE_LIKES":{modifies $LikesLinks(*p2~*p3) [dts=*p1]}

"CREATE_POST":{adds m:Post >> Messages(*p2) [ID=*p2]
    adds-ref m -hasCreator-> $Persons(*p4)
    adds-ref m -isLocatedIn-> $Countries(*p5)
    adds-ref m -container-> $Forums(*p6)
    adds-refs m -hasTag-> $Tags(*p7+)
    adds-ref model -ownedPosts-> m}

"DELETE_POST":{modifies $Messages(*p2) [dts=*p1]}

"CREATE_COMMENT":{adds c:Comment >> Messages(*p2) [ID=*p2]
    adds-ref c -hasCreator-> $Persons(*p3)
    adds-ref c -isLocatedIn-> $Countries(*p4)
    adds-ref c -replyOf-> $Messages(*p5)
    adds-refs c -hasTag-> $Tags(*p6+)
    adds-ref model -ownedComments-> c}

"DELETE_COMMENT":{modifies $Messages(*p2) [dts=*p1]}
```

```
"CREATE_KNOWS":{adds kL:KnowsLink >> KnowsLinks(*p2~*p3)
    adds-ref $Persons(*p2) -knows-> kL
    adds-ref kL -knows-> $Persons(*p3)
    adds-ref model -ownedKnowsLinks-> kL}

"DELETE_KNOWS":{modifies $KnowsLinks(*p2~*p3) [dts=*p1]}
```

Listing B.8: The mapping from events to model modifications for the SNB

B.3.2 *Queries in ITQL*

The queries IC4 and IC5 are described in Section 5.3. Their corresponding specification in ITQL is shown in Listing B.10 and Listing B.9. The specifications refer to entity types in the metamodel in Figure 5.4.

```
import 'http://mdelab.de/intempo/examples/ldbc_snb/1.0'

($p1,!(!E$p11 AND !E($p12, E$p121))
    AND (true S[0,1m] E ($p13, (!(true S[0,1]oo E$p131)))))

declarations{
    p1{p2:Person
        p2 -hasCreated-> m
        m:Post
        m -container-> f
        f:Forum}

    p11{p2:Person
        p1:Person
        kL12:KnowsLink
        p1 -knows-> kL12
        kL12 -knows-> p2
        [OCL:"p1.ID=181"]}

    p12{p2:Person
        p3:Person
        kL32:KnowsLink
        p3 -knows-> kL32
        kL32 -knows-> p2}

    p121{p1:Person
        p3:Person
        kL13:KnowsLink
        p1 -knows-> kL13
        kL13 -knows-> p3
        [OCL:"p1.ID=181"]}

    p13{p2:Person
        Mem:HasMemberLink
        f:Forum
        Mem -forum-> f
        Mem -person-> p2}

    p131{Mem:HasMemberLink}
}
```

Listing B.9: The query IC5 in ITQL

```
import 'http://mdelab.de/intempo/examples/ldbc_snb/1.0'

($q1, E($q11, E($q111, !(true S[0,1]oo E $q1111))
        AND (true S[0,1y] E($q112,
        !(true S[0,1]oo E $q1121) AND !(true S[0,*]co E($q1122, E $q11221)))))

declarations{
    q1{t:Tag}

    q11{t:Tag
        p2:Person}

    q111{p2:Person
        p1:Person
        kL:KnowsLink
        p1 -knows-> kL
        kL -knows-> p2
        [OCL:"p1.ID=181"]}

    q1111{kL:KnowsLink}

    q112{t:Tag
        m:Post
        p2:Person
        m -hasTag-> t
        p2 -hasCreated-> m}

    q1121{p2:Person
        p2 -hasCreated-> m
        m:Post}

    q1122{p1:Person
        kL2:KnowsLink
        p1 -knows-> kL2
        kL2 -knows-> p3
        p3:Person
        m2:Post
        p3 -hasCreated-> m2
        t:Tag
        m2 -hasTag-> t}

    q11221{p1:Person
        p3:Person
        kL2:KnowsLink
        p1 -knows-> kL2
        kL2 -knows-> p3
        [OCL:"p1.ID=181"]}
}
```

Listing B.10: The query IC4 in ITQL

C PUBLICATIONS

Parts of the contributions in this thesis have been published in the following peer-reviewed conference proceedings and journals.

- Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. "Incremental Execution of Temporal Graph Queries over Runtime Models with History and Its Applications." In: *Software and Systems Modeling* 21.5 (Oct. 1, 2022), pp. 1789–1829. ISSN: 1619-1374. DOI: 10.1007/s10270-021-00950-6
- Lucas Sakizloglou, Matthias Barkowsky, and Holger Giese. "Keeping Pace with the History of Evolving Runtime Models." In: *Fundamental Approaches to Software Engineering*. Ed. by Esther Guerra and Mariëlle Stoelinga. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 262–268. ISBN: 978-3-030-71500-7. DOI: 10.1007/978-3-030-71500-7_13
- Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. "A Scalable Querying Scheme for Memory-Efficient Runtime Models with History." In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. New York, NY, USA: Association for Computing Machinery, Oct. 18, 2020, pp. 175–186. ISBN: 978-1-4503-7019-6. DOI: 10.1145/3365438.3410961
- Lucas Sakizloglou, Sona Ghahremani, Thomas Brand, Matthias Barkowsky, and Holger Giese. "Towards Highly Scalable Runtime Models with History." In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '20. New York, NY, USA: Association for Computing Machinery, June 29, 2020, pp. 188–194. ISBN: 978-1-4503-7962-5. DOI: 10.1145/3387939.3388614

Our contributions are partly founded on the Metric Temporal Graph Logic (see Section 2.3) and related formal results. These foundations constitute joint work of the author and have been published in the following peer-reviewed conference proceedings and journals.

- Sven Schneider, Maria Maximova, Lucas Sakizloglou, and Holger Giese. "Formal Testing of Timed Graph Transformation Systems Using Metric Temporal Graph Logic." In: *International Journal on Software Tools for Technology Transfer* 23.3 (June 2021), pp. 411–488. ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-020-00585-w
- Sven Schneider, Lucas Sakizloglou, Maria Maximova, and Holger Giese. "Optimistic and Pessimistic On-the-fly Analysis for Metric Temporal Graph Logic." In: *Graph Transformation*. Ed. by Fabio Gadducci and Timo Kehrer.

Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 276–294. ISBN: 978-3-030-51372-6. DOI: 10.1007/978-3-030-51372-6_16

- Holger Giese, Maria Maximova, Lucas Sakizloglou, and Sven Schneider. "Metric Temporal Graph Logic over Typed Attributed Graphs." In: *Fundamental Approaches to Software Engineering*. Ed. by Reiner Hähnle and Wil van der Aalst. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 282–298. ISBN: 978-3-030-16722-6. DOI: 10.1007/978-3-030-16722-6_16

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

AC	Application Condition
AMR	Amalgamated Marking Rule
BMR	Basic Marking Rule
CEP	Complex Event Processing
E2P	Events-to-Patterns Specification Language
EMF	Eclipse Modeling Framework
EOL	Epsilon Object Language
GDN	Generalized Discrimination Network
GT	Graph Transformation
GTS	Graph Transformation System
ITQL	InTempo Query Language
LHS	Left-hand Side
MDE	Model-driven Engineering
MFOTL	Metric First-Order Temporal Logic
MTGC	Metric Temporal Graph Condition
MTGL	Metric Temporal Graph Logic
MTL	Metric Temporal Logic
NGC	Nested Graph Condition
OCL	Object Constraint Language
RHS	Right-hand Side

RTM	Runtime Model
RTM^{H}	Runtime Model with History
RV	Runtime Verification
SAS	Self-adaptive System
SD	Story Diagram
SHS	Smart Healthcare System
SNB	Social Network Benchmark
TGDN	Temporal Generalized Discrimination Network
TSSD	Timed Scenario Story Diagram
TVGDN	Temporal Validity Generalized Discrimination Network

INDEX