

A Scalable Querying Scheme for Memory-efficient Runtime Models with History

Lucas Sakizoglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese

<first-name>.<last-name>@hpi.de

Hasso Plattner Institute, University of Potsdam, Germany

ABSTRACT

Runtime models provide a snapshot of a system at runtime at a desired level of abstraction. Via a causal connection to the modeled system and by employing model-driven engineering techniques, runtime models support schemes for (runtime) adaptation where data from previous snapshots facilitate more informed decisions. Nevertheless, although runtime models and model-based adaptation techniques have been the focus of extensive research, schemes that treat the evolution of the model over time as a first-class citizen have only lately received attention. Consequently, there is a lack of sophisticated technology for such runtime models with history.

We present a querying scheme where the integration of temporal requirements with incremental model queries enables scalable querying for runtime models with history. Moreover, our scheme provides for a memory-efficient storage of such models. By integrating these two features into an adaptation loop, we enable efficient history-aware self-adaptation via runtime models, of which we present an implementation.

KEYWORDS

runtime models, model queries, temporal requirements, temporal logic, self-adaptation, incremental pattern matching

1 INTRODUCTION

A runtime model provides a view on a running system at a desired level of abstraction that can be used for monitoring, analyzing, or adapting the system through a causal connection between the model and the system [58], i.e., any relevant change of the system is reflected in the model and vice versa [13]. Runtime models typically capture snapshot-based representations of the modeled system in its current state [8]. Thereby, they provide an abstract view on a current system configuration that, via causal connection and employing model-driven engineering techniques, can support online (model-based) adaptation schemes [13, 28] which mitigate the difficulty of managing complex interconnected systems [44].

Capturing the evolution of runtime models [1] has been shown to be a promising direction to cope with the increasing complexity of software systems and their dynamic environments [9]. Furthermore, it is often desired, and sometimes required by application domains, e.g., healthcare [17], that model-based schemes recollect previous observations and utilize these *historical data* in future activities: For instance, more informed adaptations can be enabled via expanding runtime models to capture the history of system changes and interactions [24], which can be utilized to address emergent circumstances [52] or predict potential future changes [45].

Although runtime models, model queries, and adaptation based on runtime model changes have been the focus of extensive research (cf. [10]), schemes that treat the evolution of the runtime

model over time, referred to as *Runtime Model with History* (RTM^H) in the following, as a first-class citizen have only lately received attention (cf. [26]). Moreover, in order to utilize the history of complex systems that operate in highly dynamic environments for online adaptations, RTM^H technology should be capable of consolidating numerous changes into the RTM^H, often arriving at a high pace [16] and in the form of events [22], as well as provide facilities for storing and querying the historical event data in a scalable manner.

Runtime models have been utilized in (self-)adaptation schemes where incremental model queries are employed to detect issues requiring system adaptation, e.g., failures, in an efficient manner (cf. [29]). However, the history of system changes is not captured and adaptation decisions are only made based on the current system state. The idea of runtime models enriched with history in the form of past event data where queries impose temporal requirements on matched patterns has been, so far, treated only preliminarily and ad hoc, e.g., by a manual translation of a single example of a restricted form that supports only past requirements [49]. In this paper, we extend the scheme envisioned in [49] and present a version which lifts the restrictions, supports complex queries with both past and future requirements, and their systematic operationalization. We extend the scheme further to support a full online adaptation cycle employing incremental model queries, which enables history-aware adaptations where real-time, efficient storage and querying for data generated by events are key.

Our contributions are as follows. First, we present a scalable online scheme for the incremental processing of pattern-based model queries which support temporal logic operators. The approach automatically maps a temporal graph logic formula to a network of simple graph sub-queries. Secondly, our scheme allows for memory-efficient RTM^H via an automated, a priori analysis of the model queries that only keeps data in the RTM^H that are necessary to evaluate the model queries correctly. By integrating these two contributions into a self-adaptation loop, we enable efficient history-aware self-adaptation via runtime models. Finally, we present an implementation of the querying scheme embedded in an adaptation loop, evaluate it on simulated real and synthetic logs from the medical domain, and compare it to a baseline acquired by a relevant state-of-the-art tool.

The rest of the paper is organized as follows. Section 2 discusses the building blocks of our scheme. An overview of the scheme and its utilization to enable history-aware self-adaptation is presented in Section 3. The incremental matching of patterns with temporal requirements is presented in Section 4, while Section 5 details the query analysis that enables memory-efficient RTM^H. We evaluate the performance of our prototypical implementation in Section 6, discuss related work in Section 7, and conclude the paper as well as discuss future work in Section 8.

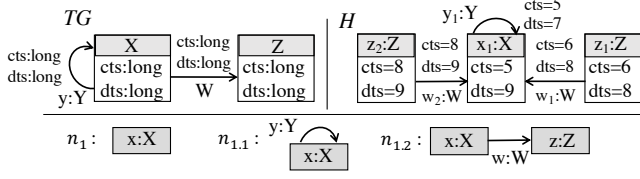


Figure 1: Type graph TG of Σ , the RTM^H H , and n patterns

2 FOUNDATIONS

2.1 Runtime Models for Self-adaptation

A *runtime model* captures a snapshot-based representation of the current state of the modeled system at a desired level of abstraction [8]. Self-adaptation can be generally achieved by adding, removing, and re-configuring components as well as connectors among components in the system architecture [42], therefore, software architecture is typically considered an appropriate abstraction level, e.g., [27, 28]. Runtime models may be used for adapting the system through a causal connection between the model and the system [58]. *Model queries* are employed to retrieve data from a runtime model. The established practice of representing the runtime model as a graph captures architectural components as vertices, connectors between components as edges, and information about the components as attributes [57]. The model conforms to a *metamodel* that specifies a language for runtime models and defines types of vertices, edges, and attributes.

Formally, a graph-based runtime model can be represented as a *typed attributed graph* where a graph is typed over a *type graph*, in the same manner a runtime model conforms to its metamodel. For an example, see Fig. 1 for the type graph TG of an abstract elementary system Σ . A graph-based representation allows for the utilization of established formalisms, such as *typed, attributed graph transformation* [23], for the maintenance and adaptation of the model, whereby *graph transformation rules* are employed to capture model queries as well as perform in-place model transformations.

In short, let G be a graph representation of the runtime model (effectively, the system architecture), and ρ a graph transformation rule. A rule ρ is characterized by a left-hand side (*LHS*) and a right-hand side (*RHS*) graph pattern which define the precondition and postcondition of an application of ρ respectively. In this context, the *LHS* describes a structural fragment of the architecture and the *RHS* the corresponding model transformation. A match m of *LHS* in G corresponds to an occurrence of *LHS* in G and identifies a part of the runtime model where the transformation should occur. The *LHS* of a rule can also be used to characterize a *graph query*, which is the equivalent graph-based notion of a model query.

To realize architectural self-adaptation, a system is equipped with a *MAPE-K* feedback loop that monitors and analyzes the system and, if needed, plans and executes an adaptation of the system via making architectural changes, i.e., adding and removing components as well as connectors among components in the system architecture [27]. All four MAPE activities are based on knowledge [39]. The feedback loop maintains a runtime model as part of its knowledge to represent the *current state* of the architecture. Rule-based self-adaptation schemes employ *adaptation rules*

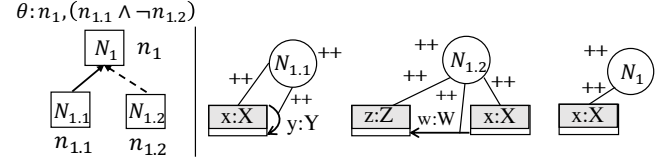


Figure 2: Example: GDN (left) and Marking Rules for θ

to capture events (during monitoring phase), check whether the events triggered any adaptation issues (during analysis) and plan for and execute an adaptation (during planning and execution respectively) [41]. The graph-based representation of runtime models allows for a realization of adaptation rules in form of graph transformation rules where analysis is performed via model queries and the system is adapted via in-place model transformations [28].

2.2 Efficient Pattern Matching for Queries

The process of finding matches of *LHS* patterns in G is called *graph pattern matching* and corresponds to the execution of a graph query specified by the pattern *LHS*. In certain cases however, simple patterns are not sufficient as a language for specifying more sophisticated *application conditions* of adaptation rules, for instance if the existence of certain model elements should be prohibited. In those cases, *LHS* patterns and thus graph queries are enhanced with a set of application conditions ac which every match m should satisfy. In the following, a graph query q is characterized by a pattern n and application conditions ac , denoted $q(n, ac)$.

A graph query is a declarative means to express a sought pattern and its application conditions. The query itself does not specify a method for its *operationalization*, i.e., instructions on how to execute the query over a graph. In this work, for the operationalization of queries, we build on [11], which supports ac formulated as *Nested Graph Conditions* (NGCs) [34]. NGCs support the nesting of patterns to bind graph elements in outer conditions and relate them to inner (nested) conditions. Moreover, NGCs support the first-order logic operators negation (\neg), existential quantification (\exists), and conjunction (\wedge) and thus have the expressive power of first-order logic on graphs [46] and constitute, as such, a natural formal foundation for pattern-based queries. The approach in [11] presents a formal framework for the decomposition of a query with an arbitrarily complex NGC as ac into a suitable ordering of simpler sub-queries, which is called a *generalized discrimination network* (GDN). A GDN is a directed acyclic graph where each graph node represents a (sub-)query. To avoid confusion, we refer to the GDN as a *network*. Dependencies between sub-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 sub-query realized by the parent node requires the presence of matches of the child node, or *negative*, i.e., the sub-query of the parent node forbids the presence of such matches. The query is executed bottom-up: the execution starts with leaves and proceeds upwards in the network. The terminal node computes the overall result of the query.

In [11], a GDN is realized as a set of graph transformation rules where each GDN node, i.e., each (sub-)query, is associated with one transformation rule. The *LHS* of the rule searches for matches of

the corresponding query in a given graph G . The *RHS* of the rule creates for each match a *marking node* in G and *marking edges* from the marking node to each element of the match—marking nodes are not to be confused with regular graph nodes in G (which, in this context, represent system components) thus we use the term *vertex* for the latter. In order to be able to create marking nodes and edges, the transformation rules of a GDN are typed over an extended type graph which adds the required types for marking nodes and edges to the initial type graph. The *LHS* of queries with dependencies have *ac* that require the existence of marking nodes of their positive dependencies and forbid the existence of marking nodes of their negative dependencies.

As an example, assume the following graph query $q(n_1, \theta)$ where $\theta := n_1, (n_{1.1} \wedge \neg n_{1.2})$, which captures the following (structural) requirement: all matches of n_1 can be extended to a pattern $n_{1.1}$ but not to a pattern $n_{1.2}$. The patterns are based on the *TG* of system Σ (introduced in Section 2.1) and are shown in Fig. 1 (bottom). The nesting of patterns implies a *binding* of the vertex of type X from inner patterns, i.e., all patterns refer to the same element in H . Note that, for presentation purposes, we adopt a simplified grammar for NGCs that omits existential quantifiers for patterns as well as the operator *true* which is always satisfied and marks the end of a nested condition, e.g., in θ , we write $n_{1.1}$ instead of $\exists(n_{1.1}, \text{true})$.

The GDN for q is shown in Fig. 2 (left), where each square represents a GDN node. Each node is associated with a marking rule. The GDN consists of three nodes, i.e., rules: the node $N_{1.1}$ for the query searching for $n_{1.1}$, the node $N_{1.2}$ for pattern $n_{1.2}$ and the topmost node N_1 for pattern n_1 . Node N_1 computes its matches by matching its pattern and checking whether both of its dependencies are satisfied (conjunction in θ). The negative dependency which captures the negation in θ (drawn by a dashed line) is satisfied when a match for N_1 can not be extended by a match for $N_{1.2}$. All nodes are realized by transformation rules whose *LHS* matches a pattern and whose *RHS* creates marking nodes and edges that mark the matches of the *LHS*. The rules for nodes $N_{1.1}$, $N_{1.2}$, N_1 are shown in Fig. 2, where (i) vertices are shown by rectangles, (ii) marking nodes by circles, and (iii) the marking nodes and edges added by a rule are annotated with "+". For presentation purposes, the illustrations of rules contain both *LHS* and *RHS*.

A GDN is capable of being executed *incrementally* and thus enables efficient, incremental pattern matching. Changes in G can propagate through the network, whose nodes only recompute their results if the change concerns them or one of their dependencies.

2.3 Runtime Models with History

A Runtime Model with History (RTM^H) [49] consolidates the evolution of a runtime model in a single graph and as such it constitutes the cornerstone of our scheme. An RTM^H can be obtained by an adjustment to the system metamodel such that each element is equipped with a *creation* and *deletion* timestamp, *cts* and *dts* respectively—see *TG* in Fig. 1. Based on monitoring data, when an element is created or deleted in the represented system, its *cts*, respectively *dts*, is updated accordingly in the RTM^H. At the time of creation, the *dts* of an element is set by default to ∞ . For an example of a RTM^H, see H in Fig. 1, where the *cts* and *dts* of each element reflects the latest monitoring data. Note that, some elements,

e.g., the edge y_1 , have been removed from the modeled system yet featured in the RTM^H. By featuring removed elements, i.e., components whose *dts* is in the past or present, an RTM^H transcends the traditional notion of causal connection.

2.4 Queries over RTM^H

We introduce graph queries that, via their *ac*, express temporal requirements on patterns. Such queries provide a powerful means to query the history or evolution of a system execution, as the latter is reflected in RTM^H. An example is the following requirement: “For all matches of n_1 in H at a time point t , at least one match of $n_{1.2}$ should be found at some time point $t' \in [t, t+2]$, that is at most 2 time units later. In addition, at each time point $t'' \in [t, t']$ in between, at least one match for $n_{1.1}$ should be present.”, where all patterns refer to the same vertex of type X in H . The specification of such requirements is enabled by *Metric Temporal Graph Logic* (MTGL) [31, 50]. MTGL builds on NGCs and *Metric Temporal Logic* [40] to enable the specification of *Metric Temporal Graph Conditions* (MTGCs) which support *metric*, i.e. interval-based, temporal operators: the future *until* (U_I , where I is a time interval¹) and *eventually* (\diamond_I), and their dual past operators *since* (S_I) and *once* (\blacklozenge_I).

MTGL reasons over a sequence of graphs, which in this context represents consecutive snapshots of the runtime model. However, as shown in [31], a graph sequence can be uniquely folded into a *graph with history*. In a graph with history, each node and edge is associated with a creation timestamp *cts* and a deletion timestamp *dts*. To store these values, the type graph is extended by appropriate attributes. MTGCs can also be equivalently checked over a graph with history, which here corresponds to an RTM^H.

The exemplary requirement above is captured by the MTGC $\zeta := n_1, (n_{1.1} U_{[0,2]} n_{1.2})$. The intuition behind *until* is reversed for the past operator *since*, e.g. $(n_{1.1} S_{[0,2]} n_{1.2})$, which requires that when $n_{1.1}$ is matched at time point t , a match for $n_{1.2}$ has existed at some time point $t' \in [t-2, t]$, and that at every time point $t'' \in (t', t]$ in between, a match for $n_{1.1}$ is present in the graph. The operators *eventually* (\diamond_I) and *once* (\blacklozenge_I) are abbreviations of *until* and *since*: $\diamond_I n_1 = \text{true} U_I n_1$ and $\blacklozenge_I n_1 = \text{true} S_I n_1$. The query $q(n_1, \zeta)$ computes all matches of n_1 in H that satisfy the MTGC ζ .

3 INTEMPO: A QUERYING SCHEME EXTENDED FOR SELF-ADAPTATION

In the following, we present the basic modules of our querying scheme named INTEMPO (from *Incremental queries with Temporal requirements*) together with the extensions that are integrated in an *adaptation engine* (Fig. 3) to realize self-adaptation based on the case-study. The engine consists of the standard MAPE activities, plus the novel and optional *maintain* activity (in gray), sensing and affecting an adaptable software via an RTM^H.

3.1 Overview of INTEMPO for Adaptation

INTEMPO consists of three basic modules. The *C* module (for construction) is executed prior to the adaptation *loop* (during *setup* of the engine) and takes a graph query with temporal requirements

¹A (time) interval I is a set $I = \{t \mid t \in \mathbb{R}_0^+ \wedge \tau \leq t \leq \tau'\}$, where $\tau, \tau' \in \mathbb{R}_0^+$, and τ, τ' is the *lower*, respectively *upper*, bound of the interval. An interval I is also denoted by $[\tau, \tau']$ and its lower and upper bound by $\ell(I)$, respectively $u(I)$.

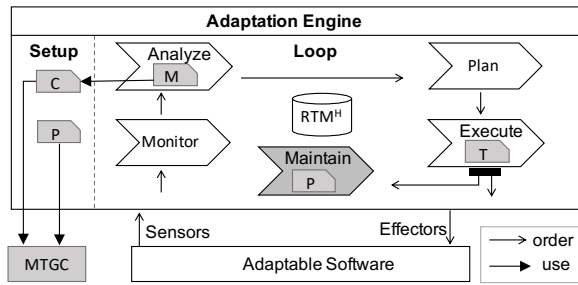


Figure 3: Overview of Adaptation Engine

captured by an MTGC as input and constructs a *temporal GDN* by decomposing the query into simpler sub-queries. The module extends the GDN construction presented in Section 2.2 by introducing concepts for capturing and handling structural matches whose validity is based on the creation and deletion time point of their elements. The *M* module (for matching) operates within the adaptation loop (cf. Fig. 3) and executes the temporal GDN, i.e., it searches for matches of queries. The module ensures that only (sub-)queries whose matches are affected by changes are re-executed and takes into account temporal requirements on matches. This module is executed in the analysis activity of the loop. The *P* module (for pruning) is executed both in setup and within the loop. In setup, it analyzes the query to derive an upper bound on the time window within which elements of the RTM^H could be used in the evaluation of the query. Then, during the maintain activity of the loop, the module uses these derived *cut-off points* to decide whether to *prune* elements from the RTM^H that have been removed from the system and are not usable by future query executions.

The previously presented modules form a (stand-alone) scalable querying scheme which enables the incremental matching of patterns with temporal requirements. The *T* module (for transformation) is an extension that enables self-adaptation via realizing the execution activity of the loop. It processes the query matches, which in this context represent issues requiring adaptation, and performs in-place model transformations, i.e., adaptation actions.

3.2 Case-Study: Smart Healthcare System

Our case-study is based on a service-based simulated Smart Healthcare System (SHS). The SHS is based on smart medical environments [48] where sensors periodically collect physiological measurements of patients, i.e., data such as temperature, heart-beat, and blood pressure, and certain medical procedures are automated and performed by devices, such as a smart pump administering medicine, based on the collected patient measurements—as otherwise a clinician would be doing. The metamodel of the SHS (Fig. 4) is influenced by the exemplar of a self-adaptive service-based medical system in [59] and captures an RTM^H as an instance of the *Architecture* class. To meet the requirements for an RTM^H , all other elements inherit from *MonitorableEntity*, i.e., are equipped with a creation and deletion timestamp.

In our SHS, services are invoked by a main service called *SHSService* to collect measurements (from patient sensors) or take medical

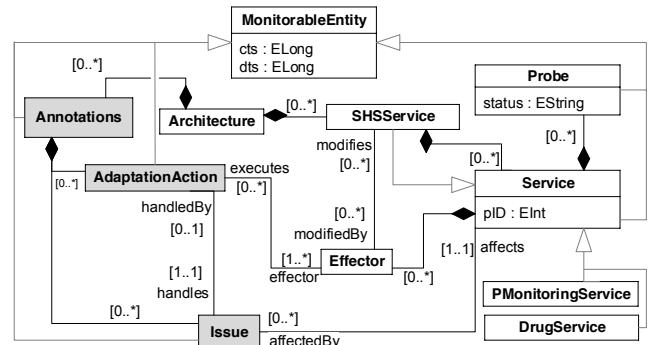


Figure 4: Metamodel of the SHS (excerpt)

actions (via patient effectors, e.g. pump), the former called *PMonitoringService* and the latter *DrugService*. Invocations are triggered by effectors (*Effector*) and invocation results are tracked via monitoring probes (*Probe*) that are attached on *Services*. *Probes* 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 *patientID* attribute which identifies the patient for whom the *Service* is invoked.

3.3 History-Aware Self-Adaptation

In the following, we build on our SHS 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 which are described below. The scenario is based on the medical guideline on the treatment of sepsis [47], a possibly life-threatening condition. We focus on the basic instruction that reads: “between *ER Sepsis Triage* and *IV Antibiotics* should be less than 1 hour”, where *ER Sepsis Triage* and *IV Antibiotics* are procedure actions for sepsis documented in real records of patients in a hospital [43]. Based on the SHS metamodel and the available hospital data, we envisage the procedure described in the guideline performed by the SHS.

In detail, an *ER Sepsis Triage* event is simulated as a *Probe* with a status equal to *sepsis*, generated for a *PMonitoringService* *pm* which has been invoked by a *SHSService* *s*. An *IV Antibiotics* event is simulated as a *Probe* with status *antibiotics* from a *Drug Service* *d* which has also been invoked by *s*. To make sure these two actions are referring to the same patient, we require that the *patientID* of *d* and *pm* are equal. The pattern fragments capturing the occurrence of these events in our SHS are depicted in patterns p_1 and $p_{1.2}$ in Fig. 5. Based on p_1 and $p_{1.2}$, the instruction is formulated in MTGL by the MTGC $\psi := (p_1, (\diamond_{[0,3600]} p_{1.2}))$, that is, for every match of p_1 which identifies a (previously untreated) patient with sepsis, eventually in the next hour there is a match for pattern $p_{1.2}$ which identifies the administration of antibiotics to that patient. The system is assumed to track time in seconds. We describe the adaptation activities in detail.

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 ,

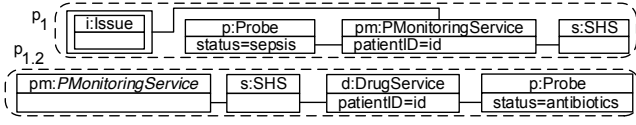


Figure 5: Illustrated Patterns for Case-study

which is an instantiation of the *Architecture*. Therefore, the RTM^H is updated to represent the current architectural system configuration enriched with the relevant temporal data.

Analyze. The analysis activity detects the potential adaptation issues, which in this context are captured by violations of ψ , i.e., the existence in the RTM^H of structural patterns that reflect sepsis cases (p_1) without associated antibiotics ($p_{1,2}$) within one hour. The detection is based on the execution of the temporal GDN by the M module of *INTEMPO*. The temporal GDN is obtained by the C module during the setup of the adaptation loop.

We remark that we aim to detect violations of ψ . Therefore, in order for matches to constitute violations of sepsis cases that can be adapted, we execute the following graph query: $q(p_1, \phi_1)$, where $\phi_1 := p_1, \neg(\diamond_{[0,3600]} p_{1,2})$. Furthermore, in order to challenge our scheme with a more complicated scenario, we also search for violations for a variation of ψ . Namely, that no patient with sepsis should be released from the medical environment prior to being treated, a requirement that resembles conformance checks of medical procedures (cf. [43]). Once more, we rely on the real hospital data and specifically the *Release* event. The structural pattern corresponding to *Release* is pattern $p_{1,1}$, a minor modification of pattern $p_{1,2}$, where the probe attached to a monitoring service has the value *release*. The requirement is captured by $\phi_2 := p_1, \neg(\neg p_{1,1} \cup_{[0,3600]} p_{1,2})$. Note that an important aspect of analysis is the handling of *potential violations*, i.e., the matching of a sepsis pattern p_1 that is not yet associated to an antibiotics pattern $p_{1,2}$ at the current RTM^H although there is still time for the requirement to be fulfilled in the future. The planning activity only detects these cases as violations if the time difference between the lower bound of the validity interval of the match and the current time point is greater than the until interval in the MTGC, which, in this case, is 3600 seconds.

The matches detected by the M module constitute adaptation issues, and similar to [29], adaptation-related classes (in gray in Fig. 4) are employed to facilitate the adaptation. During analysis, the monitoring service involved in detected issues is annotated with an instance of the *Issue* class. Therefore, to ensure that only new violations are matched, p_1 contains an *Issue* node i (Fig. 5) surrounded by a box which designates a negative pattern that should not be matched. Issue nodes and other adaptation-related classes are created by ordinary transformation rules.

Plan and Execute. In planning, the engine searches for sepsis probes annotated with an issue. Upon finding them, it attaches an *Effector* on the service to which the probe is attached. In execution, the T module searches for effectors and upon finding them takes an adaptation action, i.e., administer antibiotics to the patient via a drug service. This adaptation action is also reflected in the RTM^H by creating an *AdaptationAction* which is associated to the handled *Issue*.

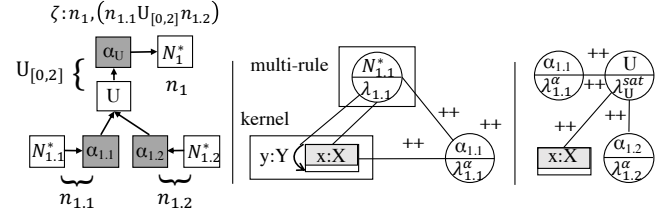


Figure 6: Temporal GDN: Network and Marking Rules for ζ

Note that the adaptation has to respect the encoding of the RTM^H , which means setting the *cts* of created elements appropriately.

Maintain. During maintenance, the P module uses the cut-off points derived after the analysis of the MTGC during setup and *prunes* the RTM^H , i.e., it removes all elements that have a *dts* in the past and cannot be used in future query evaluations. Following the removal of elements, the GDN is re-executed to update matches.

4 INCREMENTAL MATCHING OF PATTERNS WITH TEMPORAL REQUIREMENTS

In this section we present the inner workings of the matching module of *INTEMPO*. The module incrementally searches for matches of a query whose *ac* is formulated in MTGL in a RTM^H . The matching relies on a decomposition of an MTGC into simpler queries based on a temporal GDN and the subsequent incremental, bottom-up execution of the latter.

4.1 Matches and Their Lifespan

In the following, we refer to the framework in [11] (see Section 2.2) as *base approach*. The base approach builds on graph queries where application conditions are formulated as NGCs but does not support the temporal operators *until* and *since* of MTGL. Building on the base approach, we present our extensions, collectively referred to as *temporal approach* or temporal GDN, which allows for the encoding of temporal operators of MTGL by a GDN, and thus enable the incremental matching of patterns with temporal requirements.

The temporal approach differs from the base approach (see Section 2.2) in two key aspects: First, vertices and edges of a RTM^H encode data on their creation and deletion (via the *cts* and *dts* attributes respectively), i.e. their *lifetime*, which introduces the notion of a *lifespan* of a match, i.e. the time span for which the lifetimes of all nodes and edges of the match overlap; Secondly, *ac* that express temporal requirements for structural patterns (enabled by MTGL) extend the notion of a (structural) match with a period of *temporal validity*. For instance, in ζ from Section 2.4, a match for n_1 is not valid (although present in the graph) unless a match for $n_{1,2}$ is found within the specified interval. Note that, although in the following we consider both vertices and edges, our implementation aligns with recent approaches and focuses on matching vertices, as, in the general case, edges can be encoded as vertices.

4.2 Marking Rules for Matches

The intersection of two intervals is always an interval, whereas the union of two intervals i_1 and i_2 can only sometimes be encoded as an interval. In this case, we say that i_1 and i_2 are *adjacent*, i.e.

$\text{adjacent}(i_1, i_2) = \exists i_U \in \mathcal{J} : i_U = i_1 \cup i_2$, where \mathcal{J} is the set of all intervals. To encode unions that result in disjoint intervals, i.e. disconnected sets of time points, we define the *fragmented interval* $\mathbb{I} = \{i \mid i \in \mathcal{J}\}$. Note that in the following, if we perform set operations on fragmented intervals, we consider the set of time points encoded by the fragmented interval rather than the intervals in \mathbb{I} . To capture the lifespan of matches, we equip the types of marking nodes in the type graph with an attribute λ of type fragmented interval. We extend the marking rules of the base approach to set the attribute λ of the created marking nodes to the lifespan of the match m , where the latter is defined as the intersection of the lifetimes of the matched elements E :

$$\lambda^m = \bigcap_{e \in E} d^e \quad (1)$$

where d^e is given by $e.\lambda$ if e is a marking node, and by $[e.cts, e.dts]$ otherwise. The functionality of the rule remains otherwise unchanged. We name this extended marking rule MR^* .

4.3 Marking Rules for Aggregating Matches

The requirements of *until* in ζ from the example in Section 2.4 stipulate that at least a single match for $n_{1,1}$ is present in the graph until at least one $n_{1,2}$ is matched. In order to evaluate this, we need to keep track of the lifespans of all matches for $n_{1,1}$ and $n_{1,2}$. The number of these matches might vary in every GDN update, a property which is not covered by conventional graph transformation rules as presented in Section 2.1. To allow for a marking node to possibly be associated with a varying number of graph elements, as required by *until* and *since*, we introduce the concept of an *amalgamated marking rule* (αMR). The latter stems from amalgamated graph transformations [12], where an arbitrary number of parallel transformations are amalgamated, i.e., merged, into a single rule applied to the same model in one transformation step. The *LHS* of αMR contains a *kernel* of graph elements that are bound by the enclosing operator (in ζ , that would be a vertex of type X) and a *multi-rule* which matches an arbitrary number of instances of a certain marking node type. An αMR thus groups the marking nodes matched by the multi-rule by matches of the kernel. Hence, the αMR corresponds to a GDN node with a single dependency to the node that creates the marking nodes which the αMR groups.

Similar to MR^* , the *RHS* of an αMR creates an α marking node which is connected to the marking nodes of its dependency (matched by the multi-rule) and the elements of its pattern in the kernel marked by those marking nodes. If the dependency is positive, the lifespan of the marking node αMR is computed by intersecting the lifespan of the match of the kernel m_K with the union of the lifespans of the marking nodes E^M matched by the multi-rule:

$$\lambda_{PAC}^\alpha = \lambda^{m_K} \cap \bigcup_{e \in E^M} \lambda^e \quad (2)$$

If the dependency is negative, the relative complement is computed instead of the intersection:

$$\lambda_{NAC}^\alpha = \lambda^{m_K} \setminus \bigcup_{e \in E^M} \lambda^e \quad (3)$$

See Fig. 6 (middle) for an example of αMR for $n_{1,1}$ from ζ . Note that the depiction of marking rules for the temporal GDN is illustrated

by a split marking node: The bottom compartment contains the lifespan of the marking node.

4.4 Marking Rules for Temporal Operators

Finally, we introduce two dedicated marking rules for *until* and *since*, named UMR and SMR respectively, whose *LHS* patterns contain elements that are bound by its enclosing query. The UMR and SMR have a dedicated dependency for both their left and right operand and perform a special computation for the lifespan of the marking nodes they create. Let $\lambda_\ell^\alpha, \lambda_r^\alpha$ be the lifespan of left, respectively right, operand of *until*. An example of the UMR for ζ is shown in Fig. 6 (right). The computation of the lifespan of UMR is the following: for every right interval $i \in \lambda_r^\alpha$ and every adjacent left interval $j \in J_i$, where $J_i = \{j \mid j \in \lambda_\ell^\alpha \wedge \text{adjacent}(i, j)\}$, we compute the *right pivot* interval ν of i and the *left pivot* interval μ of j .

$$\nu(i) = [\ell(i) - u(I_U), u(i) - \ell(I_U)] \quad (4)$$

$$\mu(j) = [\ell(j), u(j) - \ell(I_U)] \quad (5)$$

Considering only adjacent intervals ensures the satisfaction of the requirement of *until* that there is a match of the left operand continuously until there is a match of the right operand. The pivot intervals allow us to check whether the matches of the left and right operand occur with appropriate timing with respect to the relative interval I_U defined by the *until* operator in the formula, i.e. the interval $[0, 2]$ in ζ . The intersection of $\nu(i)$ and $\mu(j)$ then marks an interval where *until* is satisfied. The total satisfaction λ_U^{sat} is computed as the union of all satisfaction intervals:

$$\lambda_U^{\text{sat}} = \bigcup_{i \in \lambda_r^\alpha} \bigcup_{j \in J_i} \nu(i) \cap \mu(j) \quad (6)$$

The computation of the lifespan for a SMR is similar to a UMR. They only differ in the computation of ν and μ which for *since* are:

$$\nu(i) = [\ell(i) + \ell(I_S), u(i) + u(I_S)] \quad (7)$$

$$\mu(j) = [\ell(j) + \ell(I_S), u(j)] \quad (8)$$

where I_S is the specific interval of the *since* operator. For the operators *eventually* and *once*, where the left-hand operand is always true, their λ_ℓ^α is equal to \mathbb{R}_0^+ .

The case where $\ell(I_U) = 0$ (or, in the case of *since*, $\ell(I_S) = 0$) is special, because according to the specification of MTGL, the formula can be satisfied without any occurrence of a match for the left operand. Therefore, the computation of λ_U^{sat} is slightly adapted:

$$\lambda_{U_0}^{\text{sat}} = \lambda_U^{\text{sat}} \cup \lambda_r^\alpha \quad (9)$$

The computation is analogously adapted for *since*.

4.5 GDN Construction and Example

In the previous section, we were concerned with marking nodes created by the GDN rules. In this section, we focus on GDN nodes which form the components of the network and represent rule applications. Regarding the construction of a temporal GDN from an MTGC, there are two extensions to the base approach: First, we represent UMR and SMR with two new types of GDN nodes for *until* and *since* respectively. For each such node, we add dependencies to the GDN nodes realizing the left and right operands of the corresponding temporal operator. Secondly, instead of adding

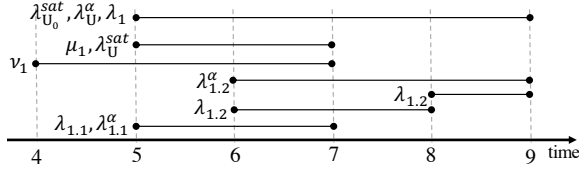


Figure 7: Computed Intervals for ζ over $\text{RTM}^H H$

direct dependencies from a GDN node β to another GDN node γ , where both β and γ realize a part of the MTGC, we construct an intermediate α node, i.e., a GDN node realized by an α MR. We add a dependency from α to γ , which is negative iff the original dependency is negative, and a positive dependency from β to α . Note that this means that in a temporal GDN, only α nodes may have negative dependencies.

See Fig. 6 (left) for the temporal GDN of ζ , where the novel α nodes are in dark gray. The patterns $n_{1,1}$ and $n_{1,2}$ are represented by nodes $N_{1,1}^*$ and $N_{1,2}^*$ respectively. These nodes are both dependencies of their respective α nodes. The U node is dependent on the two α nodes. Finally, node N_1^* , the one created for n_1 , is dependent on an α node with dependency U. For more complex constructions, the instructions in [11] apply. We now present an example of an execution of the query $q(n_1, \zeta)$ over the $\text{RTM}^H H$ in Fig. 1. In the example, we assume the query is executed at time point $t = 9$. The execution consists of seven phases. The computed intervals are illustrated in Fig. 7.

- I An MR* rule for $n_{1,1}$ is applied (node $N_{1,1}^*$ in the GDN in Fig. 6). One match is found and thus one marking node is created with $\lambda_{1,1} = [5, 7]$, i.e., the lifespan of the match, based on Equation 1.
- II The α MR of node $\alpha_{1,1}$ is applied. One vertex of type X is matched and one marking node of type $N_{1,1}^*$. The marking node created by α MR groups the lifespans of its dependencies (in this case, only one equal to $[5, 7]$) based on Equation 2 and stores the result into its attribute $\lambda_{1,1}^\alpha$.
- III The MR* of node $N_{1,2}^*$ is applied. Two matches are found and two marking nodes are created with lifespans $[6, 8]$ and $[8, 9]$ respectively.
- IV The α MR of node $\alpha_{1,2}$ is applied. One match of type X is found and two marking nodes of type $N_{1,2}^*$. One marking node is created whose result groups the lifespans of its dependencies and stores it in $\lambda_{1,2}^\alpha = [6, 9]$.
- V The UMR of node U is applied. One match for a vertex of type X is found and two marking nodes for the left and right operand respectively. The lifespans of the right marking node $\lambda_{1,2}^\alpha$ are checked on whether they are adjacent to any of the intervals in the lifespan of the left marking node $\lambda_{1,1}^\alpha$. This is true for two lifespans. The right pivot $v_1 = [4, 7]$ and left pivot $\mu_1 = [5, 7]$ for these two lifespans are computed based on Equation 4 and Equation 5 respectively. The lower bound of *until* is 0 so we have the special case where the satisfaction interval of *until* is computed by Equation 9 as the union between the intersections of μ and v and the $\lambda_{1,2}^\alpha$ which is $[5, 9]$.
- VI The α MR of node α_U is applied. The lifespan of the created marking node is computed to be $\lambda_U^\alpha = [5, 9]$.

VII The MR* of node N_1^* is applied. One vertex of type X is matched and one marking node of type α_U . The lifespan of the match is their intersection: $[5, 9]$.

Finally, the executed query returns a match for n_1 and the (possibly fragmented) interval during which, besides being structurally present in the graph, the match satisfies the temporal requirements expressed by the MTGC ζ . Employed for self-adaptation, the temporal GDN incrementally computes in each execution step all new matches, i.e., adaptation issues, *as soon as possible* and marks them with a marking node. These marking nodes, together with the graph elements they mark, remain in the RTM^H in subsequent steps.

5 MEMORY-EFFICIENT RTM^H

In this section we present the query analysis of the P module that allows for memory-efficient RTM^H . By deletion timestamps, an RTM^H retains information about elements that have been possibly removed from the represented system. This wealth in insight comes with a price: the perpetual accumulation of historical data causes the RTM^H to constantly grow in size. A possible remedy is to utilize external, domain-specific retention policies of patient records, such as the ones publicly available for national healthcare systems, e.g., [53]. Based on such policies, a process can perform periodical *garbage-collection*, where obsolete elements are pruned from the model and thus memory is freed.

Although such generic removal policies provide a certain upper bound for memory consumption, cluttering the model with obsolete data may lead to deteriorating performance of the pattern matching as more elements have to be considered. A more fine-grained solution than generic policies is to derive this information by the considered queries and their temporal requirements.

We define a function that computes a cut-off point for elements in the RTM^H based on an MTGC χ_1 as follows:

$$\kappa(\chi_1) = \begin{cases} t' + \max(\kappa(\chi_{1,1}), \kappa(\chi_{1,2})) & \text{if } \chi_1 = \chi_{1,1}U_{[t,t']}\chi_{1,2} \\ t' + \max(\kappa(\chi_{1,1}), \kappa(\chi_{1,2})) & \text{if } \chi_1 = \chi_{1,1}S_{[t,t']}\chi_{1,2} \\ \max(\kappa(\chi_{1,1}), \kappa(\chi_{1,2})) & \text{if } \chi_1 = \chi_{1,1} \wedge \chi_{1,2} \\ \kappa(\chi_{1,1}) & \text{if } \chi_1 = \neg \chi_{1,1} \\ \kappa(\chi_{1,1}) & \text{if } \chi_1 = \exists(n_1, \chi_{1,1}) \\ 0 & \text{if } \chi_1 = \text{true} \end{cases} \quad (10)$$

Recall that all MTGCs reduce to *true* (Section 2.4) but for presentation purposes, their syntax has been simplified. Moreover, here we assume that cut-off points need to be calculated for only one query. In case multiple queries are executed at once over the RTM^H , Equation 10 has to be adjusted to calculate the upper bound based on all queries in question.

The cut-off point $\kappa(\chi_1)$ corresponds to an upper bound for the maximum number of time units after which a deleted graph element can still be part of a match that may contribute to the satisfaction of the formula χ_1 at the time of checking it. For each graph element e , we can hence derive an upper bound of its *relevance window*, i.e. the window during which an element could be used in checking the *ac* of queries, by $t_{max}^e = e.dts + \kappa(\chi_1)$. By deriving the maximum time point that deleted elements can be reused in checking the *ac*, we ensure that no element has been pruned prior to the end of its relevance window, and thus that completeness of query results is

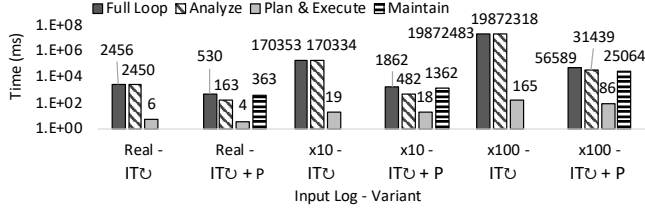


Figure 8: Cumulative Time of Loop Activities for ϕ_1

not affected. For the formula ζ from Section 2.4, the derived cut-off is 2, i.e., elements need to be retained in the RTM^H H for 2 time units after their deletion.

Pruning could reduce the size of the matching search space and thus improve the matching time. On the other hand, due to the re-computation of matches after one or more removals have occurred, pruning could potentially incur an increase in the overall adaptation time. Provided such considerations have been made, this step renders an RTM^H memory-efficient, in that, it is sustainable regardless of whether external memory-saving measures are present or too coarse, e.g., the ones in healthcare mentioned above.

6 EVALUATION

Our implementation of INTEMPO embedded in the adaptation engine (cf. Section 3) is based on the Eclipse Modeling Framework (EMF) [25, 54], which is a widespread MDE technology for creating metamodels of architectures. For pattern matching, we employ the *Story Diagram Interpreter* (SD) [30] optimized according to [2]. SD uses *local search* to start the search from a specific element of the graph and thus reduces the pattern matching effort [38]. For computations on intervals we employ an open-source library [33]. For the removal of elements from the runtime model, 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 scalable.

To evaluate our implementation, we developed a simulator of the adaptable SHS presented in Section 3.2. Our simulations replay events, based on real as well as synthesized patient data on a RTM^H . The logs are described in Section 6.1. Based on the processed log event, a corresponding structural fragment is added to the RTM^H , for instance, an *ER Sepsis Triage* corresponds to the pattern p_1 (Fig. 5) being added to the model. We implemented two variants of INTEMPO: IT^\cup which contains the construction (C), the matching (M) and the transformation (T) module and $\text{IT}^\cup + P$ which contains all of the above plus the pruning (P) module—the left circle arrow symbolizes the loop-based adaptation. In Section 6.2, we compare the time-scalability of the two variants based on multiple logs with a varying arrival rate of events.² We compare the performance of IT^\cup and $\text{IT}^\cup + P$ in the detection of adaptation issues, i.e., analysis activity, to a baseline acquired by **MONPOLY**, a state-of-the-art event-based monitoring tool. Besides time- and memory-scalability, the comparison touches on aspects of *usability* and *monitorability*. Finally, we discuss threats to validity in Section 6.4.

²All experiments and simulations have been conducted on a QuadCore Intel i7 and an OpenJDK8 JVM. Memory measurements are based on values reported by the JVM.

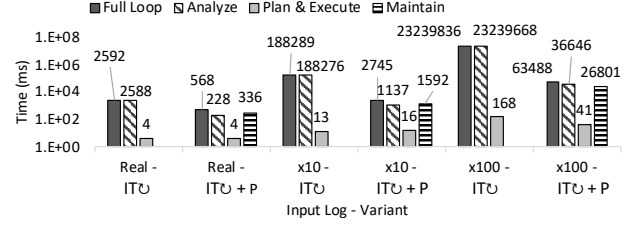


Figure 9: Cumulative Time of Loop Activities for ϕ_2

6.1 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 [43]. Each trajectory comprises a *sequence* of events, among which, we are interested in the *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 (as an ER initiates a trajectory). We use statistical probability distribution fitting to find the best-fitting distribution that characterizes the inter-arrival times between: two ER events (IAT_E), an ER and an IV (IAT_I), and an ER and an RE (IAT_R). Then, we use statistical bootstrapping [19] to generate two synthetic logs, *x10* and *x100*, with IAT_E values that are 10 and 100 times smaller respectively than IAT_E values of the *real* log, while IAT_I and IAT_R 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.) 10 and 100 times respectively, allowing us to test the scalability of INTEMPO without compromising the statistical characteristics of the *real* log.

6.2 Implementation Variants IT^\cup and $\text{IT}^\cup + P$

Although pruning the RTM^H is required for memory-efficiency, we implemented a variant of INTEMPO without the maintain activity, i.e., without pruning. Besides serving as a baseline for $\text{IT}^\cup + P$, IT^\cup could be useful in application domains where it is known that queries of interest change often and thus cut-off points cannot be derived a priori, as historical data might be useful for another query in the future. Moreover, in certain cases, the incurred cost of pruning on the loop execution time might be undesirable.

We evaluate IT^\cup and $\text{IT}^\cup + P$ with respect to their *reaction time* (or loop time) over an increasing amount of log events and model sizes. In this context, the reaction time is equal to the required time for a 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 analysis, planning, execution and, for $\text{IT}^\cup + P$, maintenance time. The time spent in monitoring, i.e., processing an event and adding the corresponding fragment to the RTM^H , is negligible and thus not measured. A loop is invoked periodically based on a predefined but modifiable frequency. In our experiments, based on the IAT_E of the logs, we set the invocation frequency to one hour, to avoid frequent invocations where there are no events to be processed. The invocation frequency coincides with the maximum delay of a violation detection, i.e., in the worst

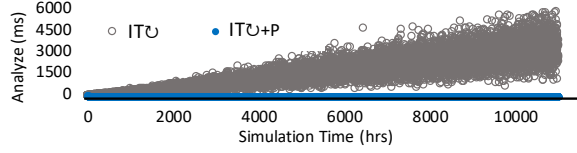


Figure 10: Analysis Time for Engine Variants (ϕ_1 - $\times 100$)

case, a violation will occur just after the loop and will be detected at the next invocation which in this case is exactly after one hour.

The C module of INTEMPO as well as the function κ (Section 5), that derives the cut-off points used by P module of $\text{ITU}+P$, are executed only once during the setup of the loop. Each experiment proceeds as follows: Events from the logs are processed and changes are made to the RTM^H ; The loop is invoked at the predefined intervals which includes the analysis activity, where the M module executes queries that search for violations of ϕ_1 and ϕ_2 (see Section 3.2) that constitute adaptation issues, and the T module performs transformations corresponding to adaptation actions; Then, for $\text{ITU}+P$, maintenance is done and matches are recomputed.

The experiments simulate the data in *real*, $\times 10$, and $\times 100$ logs. Each experiment entails the execution of one variant measured for one performance aspect (time or memory). Fig. 8 and Fig. 9 depict the cumulative time (in logarithmic scale) for each of the measured loop activities and the reaction, i.e. total, time for ϕ_1 and ϕ_2 . As expected, the results are mainly influenced by the analysis activity, which is when issues are detected. Two parameters in the conducted experiments increase simultaneously: first, the number of processed events (total number and per loop) and, second, the size of the RTM^H over which the queries are executed. The analysis time of ITU increases with respect to these two parameters but at a smaller pace. Thanks to pruning, $\text{ITU}+P$'s analysis time increases yet at a considerably smaller pace compared to ITU . However, since pruning forces a re-computation of the results, the time it requires is non-negligible. The analysis time for each loop of the two variants for the $\times 100$ log is shown in Fig. 10. The pruning of RTM^H allows the analysis time of $\text{ITU}+P$ to remain constant.

6.3 Comparison to State-of-the-art Tool

During the analysis activity, INTEMPO processes a sequence of events which represents an ongoing system execution and checks whether the observed sequence (captured in the RTM^H) satisfies a formal specification (captured by one or more MTGCs). This monitoring approach is also known as Runtime Monitoring (RM) [3] and we therefore employ a prominent RM tool in order to acquire a baseline for the performance of ITU and $\text{ITU}+P$ in detecting issues during analysis. We compare to MONPOLY [5, 6], a mature

Table 1: Memory Consumption (max) for ϕ_1 (MB)

	<i>real</i>	$\times 10$	$\times 100$
ITU	43	174	1544
$\text{ITU}+P$	29	30	33
MONPOLY	20	31	165

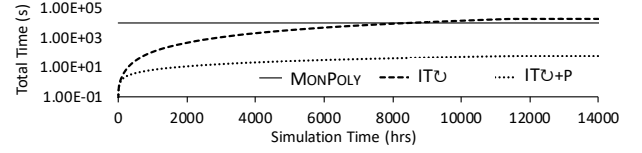


Figure 11: Total Experiment Time vs. MONPOLY for ϕ_1

command-line tool which notably combines an adequately expressive specification language with an efficient incremental monitoring algorithm and has been the reference point in evaluations of other RM tools [20, 36] and among top-performers in an RM competition [4]. Its specification language is the Metric First-Order Temporal Logic [5] (MFOTL) which employs first-order *relations* to capture system entities and their relationships. The usage of a temporal logic facilitates the translation of temporal requirements between MONPOLY and our implementation. For the encoding of the SHS metamodel by relations we translate each edge following standard practices (cf. [46]) and each vertex into a relation with an arity that depends on the number of class attributes. Based on this encoding, we translate each log event into a series of relations.

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 build an MFOTL formula that would consider all possible events 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.³ Leveraging the knowledge of the actual order in which events occur in the simulation, we simplify the formulas for MONPOLY by formulating only the correct ordering. This creates an advantage for MONPOLY in the comparison with our implementation. The difficulties in emulating pattern matching with MONPOLY indicate the tool is sub-optimal for graph-based models and pattern matching. We map ϕ_1 in a straightforward manner to its MFOTL equivalent, i.e., the temporal operators remain intact and relation fragments are used instead of patterns. This is not possible for ϕ_2 , 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 $p_{1.1}$ at a given time point when it does not exist is satisfied by infinite values and thus non-monitorable. In the following, we compare to MONPOLY only for ϕ_1 .

We acquire the baseline by executing MONPOLY only once at the end of each simulation. The latest MONPOLY version (1.1.9) was used and run on the same machine as the implementation variants. The results for the execution time (in seconds) for $\times 100$, which emphasizes the trends of smaller logs, are shown in Fig. 11. The results are compared to the *total experiment time* (not only analysis time) of our variants as other adaptation-related activities could affect their analysis times. Issue detection with MONPOLY is faster than ITU , however, $\text{ITU}+P$, due to pruning, outperforms MONPOLY . Table 1 shows similar results for memory consumption.

³Since MONPOLY outputs the time point of a violation, forward-looking matching, i.e., matching a relation in the past and subsequently searching for other relations in its future, would not produce the desired result as it would always only output the time point the first violating relation was matched.

6.4 Threats to Validity

Threats to internal validity concern the experimental setting. We systematically evaluated $IT\cup$, $IT\cup+P$, and $MONPOLY$ by using a controlled simulation of an SHS. Our focus was the effects of incremental pattern matching and pruning on the time- and memory-scalability of the variants measured during the analysis activity of the adaptation loop. To solely focus on these effects, the two variants share identical monitoring, planning, and execution activities and they use the same architectural metamodel. $MONPOLY$ is evaluated only on the analysis. The experiments draw from an example where instructions are deterministic. Moreover, the experiments simulate multiple input logs with different properties to allow for testing the variants in different circumstances such as increasing system load. The logs used are either real data or data extracted from real data employing sophisticated statistical bootstrapping.

Threats to external validity may restrict the generalization of our evaluation results outside the scope of our experiments. We evaluated our querying scheme on simulated real and synthetic data while enacting an instruction from a real medical guideline [47]. Our SHS metamodel is influenced by a peer-reviewed self-adaptation artifact. As a result, we have confidence that our evaluation, to a certain extent, holds for real scenarios. While our experiments can serve as an indication to the scalability of our querying scheme, quantitative claims on scalability require more extensive simulation scenarios taking into consideration real IoT device properties, such as memory. $MONPOLY$ is not built for pattern matching and our emulation of the latter might have room for improvement. $MONPOLY$'s semantics is *point-based* while the semantics of MTGL (and thus of our implementation) is interval-based. This fundamental difference did not impact our experiments but might influence more extensive comparisons. For the property that $MONPOLY$ could not monitor, there might exist equivalent, monitorable MFOTL formulas which, however, would not correspond to the MTGC straightforwardly.

7 RELATED WORK

The efficient storage of historical data from the system execution in a (graph-based) model has been the focus of extensive research, e.g. temporal graphs [35], the same is not true however for the scalable querying and the sustainability of runtime models [10]. Recent works build on a database, either graph- [26] or map-based [32] to store model versions which are queried by means of an OCL extension that supports temporal primitives. Neither [26] nor [32] consider an online setting where query matches can be utilized while the system is running. In an online setting, storing multiple versions of the model and accesses to a database storage take a significant toll on real-time querying performance (the latter indicated by the evaluation in [32]), especially for far-reaching past queries. To improve performance, the authors in [26] provide the capability to a priori manually annotate such queries, such that their matches are pre-computed while the system evolves, which, however, is automatically achieved by the temporal GDN of $INTEMPO$. Moreover, $INTEMPO$ uses an in-memory representation of the model, as shared memory space generally makes the real-time querying faster, which is key for the online setting and the (adaptive) systems of interest in this work. Finally, a solution for the perpetual accumulation of historical data is missing from both [26] and [32].

The setting of our case-study resembles *streaming* [18] and *active model transformations* [7] where the model and query results are assumed to be continuously updated by a stream of model elements or events that are mapped to model elements. These paradigms increase demands on the performance of the pattern matching, which previous approaches have met via employing incremental query evaluation frameworks [56] (similarly to $INTEMPO$) and the distribution of pattern matching [55]. The approach in [21, 22] generates events when patterns are matched and then employs *complex event processing* to check whether generated events occur within a given time window, thus capturing, albeit compositely and to a certain extent, temporal requirements on matched patterns. Contrary to these approaches, $INTEMPO$ natively encompasses the history of model evolution in the model representation, the query specification language, as well as during pattern matching.

As previously mentioned, our setting is also related to Runtime Monitoring (see Section 6.3). Besides $MONPOLY$ however, other approaches provide no or only partial support for key features of $INTEMPO$ such as events containing data, temporal requirements, or metric temporal operators: The work in [20] concerns propositional events, i.e., containing no data, and is thus unsuitable for the use-cases discussed in this paper; In [14, 15] an event-based scheme for the incremental matching of graph patterns in a runtime model is presented which however does not support the integration of model queries and temporal requirements; the tool in [36, 37] employs relations and discards unusable data (similar to pruning) but its logic supports only past operators without intervals.

In [49] we presented a preliminary version of $INTEMPO$ that is based on an ad hoc, manual translation of a single, syntactically restricted past MTGC that does not account for the aggregation of matches and also presents an ad hoc, manual derivation of cut-off points. The $INTEMPO$ version presented here translates MTGCs and derives cut-off points automatically. Moreover, it introduces the required concepts and facilities for aggregating matches, supports both past and future operators, and enables a complete adaptation loop. In [51] we presented the formal foundation for the translation of a syntactically restricted MTGC to an NGC which is then checked against an event-based execution aggregated in an RTM^H -like graph. However, the approach does not consider model queries, rather a non-incremental satisfaction check of the MTGC, nor does it consider past operators or a means to limit data accumulation.

8 CONCLUSION AND FUTURE WORK

We have introduced a querying scheme where graph-based model queries are integrated with temporal requirements on patterns, formulated in a temporal graph logic. Our scheme enables the incremental execution of queries over runtime models with history. Building on self-adaptive systems, in our case-study query matches capture adaptation issues in the runtime model which are handled by in-place model transformations. Our scheme offers the option to retain in the model only information that are relevant to the query executions. We present an implementation which we evaluate based on a simulation of both real as well as synthetic data and compare its efficiency in detecting issues to a relevant monitoring tool.

As future work, we plan to present a formalization of our approach, integrate more sophisticated decision-making schemes in

the planning phase, improve the performance of our implementation by employing indexing structures that can index matches based on their intervals, and evaluate the performance of other incremental query evaluation frameworks such as RETE networks.

ACKNOWLEDGMENTS

This work is partially supported by the German Research Foundation (DFG) under GI 765/8-1.

REFERENCES

- [1] Alessandro Artale, Christine Parent, and Stefano Spaccapietra. 2007. Evolving objects in temporal information systems. *Annals of Mathematics and Artificial Intelligence* 50, 1-2 (2007), 5–38.
- [2] Matthias Barkowsky and Holger Giese. 2020. Hybrid search plan generation for generalized graph pattern matching. *J. Log. Algebraic Methods Program.* 114 (2020), 100563. <https://doi.org/10.1016/j.jlmp.2020.100563>
- [3] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios E. Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. 2018. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). Lecture Notes in Computer Science, Vol. 10457. Springer, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5
- [4] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Deckert, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. 2019. First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *STTT* 21, 1 (2019), 31–70. <https://doi.org/10.1007/s10009-017-0454-5>
- [5] David Basin, Felix Klaedtke, Samuel MÅijller, and Eugen ZÅclinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2 (May 2015), 1–45. <https://doi.org/10.1145/2699444>
- [6] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. 2017. The MonPoly Monitoring Tool. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA (Kalpa Publications in Computing, Vol. 3)*, Giles Reger and Klaus Havelund (Eds.). EasyChair, 19–28. <http://www.easychair.org/publications/paper/62MC>
- [7] Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2010. Active operations on collections. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 91–105.
- [8] Nelly Bencomo, Robert B France, Betty HC Cheng, and Uwe ÅBmann. 2014. *Models@run.time: foundations, applications, and roadmaps*. Vol. 8378. Springer.
- [9] Nelly Bencomo, Sebastian Götzt, and Hui Song. 2019. Models@run.time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling* (2019).
- [10] Nelly Bencomo, Sebastian Götzt, and Hui Song. 2019. Models@run.time: a guided tour of the state of the art and research challenges. *Software and Systems Modeling* 18, 5 (2019), 3049–3082. <https://doi.org/10.1007/s10270-018-00712-x>
- [11] Thomas Beyhl, Dominique Blouin, Holger Giese, and Leen Lambers. 2016. On the Operationalization of Graph Queries with Generalized Discrimination Networks. In *Graph Transformation - 9th International Conference, ICGT 2016, Vienna, Austria, July 5-6, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9761)*, Rachid Echahed and Mark Minas (Eds.). Springer, 170–186. https://doi.org/10.1007/978-3-319-40530-8_11
- [12] Enrico Biermann, Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Gabriele Taentzer. 2010. 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 (Lecture Notes in Computer Science, Vol. 5765)*, Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel (Eds.). Springer, 121–140. https://doi.org/10.1007/978-3-642-17322-6_7
- [13] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@run.time. *Computer* 42, 10 (2009), 22–27. <https://doi.org/10.1109/MC.2009.326>
- [14] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel VarrÅs. 2020. Distributed graph queries over models@run.time for runtime monitoring of cyber-physical systems. *Int. J. Softw. Tools Technol. Transf.* 22, 1 (2020), 79–102. <https://doi.org/10.1007/s10009-019-00531-5>
- [15] MÅarton BÅzr, GÅqbor SzilÅagyí, AndrÅas VÅúrÅtűs, and DÅqniel VarrÅs. 2018. Distributed Graph Queries for Runtime Monitoring of Cyber-Physical Systems. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*. Springer, Cham, 111–128. https://doi.org/10.1007/978-3-319-89363-1_7
- [16] Luca Catarinucci, Danilo De Donno, Luca Mainetti, Luca Palano, Luigi Patrono, Maria Laura Stefanizzi, and Luciano Tarricone. 2015. An IoT-Aware Architecture for Smart Healthcare Systems. *IEEE Internet of Things Journal* 2, 6 (2015), 515–526. <https://doi.org/10.1109/JIOT.2015.2417684>
- [17] Carlo Combi, Mauro Gambini, Sara Migliorini, and Roberto Posenato. 2012. Modelling Temporal, Data-centric Medical Processes. In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium (IHI '12)*. ACM, New York, NY, USA, 141–150. <https://doi.org/10.1145/2110363.2110382> event-place: Miami, Florida, USA.
- [18] Jesús Sánchez Cuadrado and Juan de Lara. 2013. Streaming Model Transformations: Scenarios, Challenges and Initial Solutions. In *Theory and Practice of Model Transformations - 6th International Conference, ICMT@STAF 2013, Budapest, Hungary, June 18-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7909)*, Keith Duddy and Gerti Kappel (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-642-38883-5_1
- [19] Thomas J DiCiccio and Bradley Efron. 1996. Bootstrap confidence intervals. *Statistical science* (1996), 189–212.
- [20] Wei Dou, Domenico Bianculli, and Lionel Briand. 2017. A Model-Driven Approach to Trace Checking of Pattern-Based Temporal Properties. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, Austin, TX, 323–333. <https://doi.org/10.1109/MODELS.2017.9>
- [21] IstvÅan DÅqvid, IstvÅan RÅqth, and DÅqniel VarrÅs. 2014. Streaming Model Transformations By Complex Event Processing. In *Model-Driven Engineering Languages and Systems (Lecture Notes in Computer Science)*. Springer, Cham, 68–83. https://doi.org/10.1007/978-3-319-11653-2_5 Citation Key Alias: 10.1007/978-3-319-11653-2_5
- [22] IstvÅan DÅqvid, IstvÅan RÅqth, and DÅqniel VarrÅs. 2018. Foundations for Streaming Model Transformations by Complex Event Processing. *Software & Systems Modeling* 17, 1 (Feb. 2018), 135–162. <https://doi.org/10.1007/s10270-016-0533-1>
- [23] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. 2004. Fundamental Theory for Typed Attributed Graph Transformation. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3256)*, Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg (Eds.). Springer, 161–177. https://doi.org/10.1007/978-3-540-30203-2_13
- [24] Naem Esfahani, Eric Yuan, Kyle R Canavera, and Sam Malek. 2016. Inferring software component interaction dependencies for adaptation support. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 10, 4 (2016), 1–32.
- [25] Eclipse Foundation. 2020. *Eclipse Modeling Framework (EMF)*. Retrieved Aug 10, 2020 from <https://www.eclipse.org/modeling/emf/>
- [26] Antonio Garcia, Nelly Bencomo, Juan Parra, and Luis H. Garcia-Paucar. 2019. Querying and annotating model histories with time-aware patterns. In *IEEE / ACM 22nd international conference on model driven engineering languages and systems (MODELS)*, Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgue no (Eds.). <https://research.aston.ac.uk/en/publications/querying-and-annotating-model-histories-with-time-aware-patterns>
- [27] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. 2009. Software Architecture-Based Self-Adaptation. In *Autonomic Computing and Networking*. Springer, 31–55. http://dx.doi.org/10.1007/978-0-387-89828-5_2
- [28] Sona Ghahremani, Holger Giese, and Thomas Vogel. 2017. Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures. In *Proceedings of the 14th International Conference on Autonomic Computing (ICAC)*.
- [29] Sona Ghahremani, Holger Giese, and Thomas Vogel. 2020. Improving Scalability and Reward of Utility-Driven Self-Healing for Large Dynamic Architectures. *ACM Trans. Auton. Adapt. Syst.* 14, 3 (February 2020). <https://doi.org/10.1145/3380965>
- [30] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. 2009. Improved Flexibility and Scalability by Interpreting Story Diagrams. *ECEASST* 18 (2009). <https://doi.org/10.14279/tuj.eceasst.18.268>
- [31] Holger Giese, Maria Maximova, Lucas Sakizloglou, and Sven Schneider. 2019. Metric Temporal Graph Logic over Typed Attributed Graphs. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, 282–298.
- [32] Abel Gómez, Jordi Cabot, and Manuel Wimmer. 2018. TemporalEMF: A Temporal Metamodeling Framework. In *Conceptual Modeling - 37th International Conference, ER 2018, Xi'an, China, October 22-25, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11157)*, Juan Trujillo, Karen C. Davis, Xiaoyong Du, Zhanhuai Li, Tok Wang Ling, Guoliang Li, and Mong-Li Lee (Eds.). Springer, 365–381. https://doi.org/10.1007/978-3-030-00847-5_26
- [33] Guava. 2020. *Google Core Libraries for Java*. Retrieved May 17, 2020 from <https://github.com/google/guava>
- [34] Annegret Habel and Karl-Heinz Pennemann. 2009. Correctness of High-Level Transformation Systems Relative to Nested Conditions. *Math. Struct. Comput. Sci.* 19, 2 (2009), 245–296.
- [35] Thomas Hartmann, François Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. 2017. 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*, Xudong He (Ed.). KSI Research Inc. and Knowledge Systems

- Institute Graduate School, 596–601. <https://doi.org/10.18293/SEKE2017-048>
- [36] Klaus Havelund and Doron Peled. 2018. Efficient Runtime Verification of First-Order Temporal Properties. In *Model Checking Software*, Maria del Mar Gallardo and Pedro Merino (Eds.). Vol. 10869. Springer International Publishing, Cham, 26–47.
- [37] Klaus Havelund, Doron Peled, and Dogan Ulus. 2017. First order temporal logic monitoring with BDDs. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 116–123. <https://doi.org/10.23919/FMCAD.2017.8102249>
- [38] Stephan Hildebrandt. 2014. *On the performance and conformance of triple graph grammar implementations*. Ph.D. Dissertation. University of Potsdam. <http://d-nb.info/1054564477>
- [39] Jeffrey O. Kephart and David Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50. <http://portal.acm.org/citation.cfm?id=642200>
- [40] Ron Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Syst.* 2, 4 (1990), 255–299.
- [41] Ivan Lanese, Antonio Bucchiarone, and Fabrizio Montesi. 2010. A Framework for Rule-Based Dynamic Adaptation. In *Proceedings of the 5th International Conference on Trustworthy Global Computing*. Springer-Verlag, Berlin, Heidelberg, 284–300.
- [42] Jeff Magee and Jeff Kramer. 1996. Dynamic Structure in Software Architectures. In *Proc. of the 4th Symposium on Foundations of Software Engineering*. ACM, 3–14. <https://doi.org/10.1145/239098.239104>
- [43] Felix Mannhardt and Daan Blinde. 2017. Analyzing the Trajectories of Patients with Sepsis using Process Mining. In *Joint Proceedings of the Radar tracks at the 18th International Working Conference on Business Process Modeling, Development and Support (BPMDS), and the 22nd International Working Conference on Evaluation and Modeling Methods for Systems Analysis and Development (EMMSAD), and the 8th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA) co-located with the 29th International Conference on Advanced Information Systems Engineering 2017 (CAiSE 2017), Essen, Germany, June 12-13, 2017 (CEUR Workshop Proceedings, Vol. 1859)*, Jens Gulden, Selmin Nurcan, Iris Reinhartz-Berger, Wided Guédria, Palash Bera, Sérgio Guerreiro, Michael Fellmann, and Matthias Weidlich (Eds.). CEUR-WS.org, 72–80. <http://ceur-ws.org/Vol-1859/bpmds-08-paper.pdf>
- [44] Gunasekaran Manogaran, R. Varatharajan, Daphne Lopez, Priyan Malarvizhi Kumar, Revathi Sundarasekar, and Chandu Thota. 2018. A new architecture of Internet of Things and big data ecosystem for secured smart healthcare monitoring and alerting system. *Future Generation Computer Systems* 82 (2018), 375–387. <http://www.sciencedirect.com/science/article/pii/S0167739X17305149>
- [45] Gabriel A. Moreno, Javier C’amara, David Garlan, and Bradley Schmerl. 2015. Proactive Self-adaptation Under Uncertainty: A Probabilistic Model Checking Approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 1–12.
- [46] Arend Rensink. 2004. Representing First-Order Logic Using Graphs. In *ICGT, Vol. 4*. Springer, 319–335.
- [47] Andrew Rhodes, Laura E Evans, Waleed Alhazzani, Mitchell M Levy, Massimo Antonelli, Ricard Ferrer, Anand Kumar, Jonathan E Sevransky, Charles L Sprung, Mark E Nunnally, et al. 2017. Surviving sepsis campaign: international guidelines for management of sepsis and septic shock: 2016. *Intensive care medicine* 43, 3 (2017), 304–377.
- [48] Patrice C Roy, Samina Raza Abidi, and Syed Sibte Raza Abidi. 2017. Monitoring Medication Adherence in Smart Environments in the Context of Patient Self-management A Knowledge-driven Approach. In *Smart Technologies in Healthcare*. CRC Press, 195–223.
- [49] Lucas Sakizloglou, Sona Ghahremani, Thomas Brand, Matthias Barkowsky, and Holger Giese. 2020. Towards Highly Scalable Runtime Models with History. In *15th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2020, Seoul, South Korea, October, 2020*. IEEE Computer Society. <https://arxiv.org/abs/2004.03727>
- [50] Sven Schneider, Maria Maximova, Lucas Sakizloglou, and Holger Giese. 2020. Formal Testing of Timed Graph Transformation Systems using Metric Temporal Graph Logic. *Int. J. Softw. Tools Technol. Transf.* (2020). To appear.
- [51] Sven Schneider, Lucas Sakizloglou, Maria Maximova, and Holger Giese. 2020. Optimistic and Pessimistic On-the-fly Analysis for Metric Temporal Graph Logic. In *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12150)*, Fabio Gadducci and Timo Kehrer (Eds.). Springer, 276–294. https://doi.org/10.1007/978-3-030-51372-6_16
- [52] Jolana Sebestyénová. 2007. Case-based reasoning in agent-based decision support system. *Acta Polytechnica Hungarica* 4, 1 (2007), 127–138.
- [53] United Kingdom National Health Service. 2020. *Records Management Code of Practice for Health and Social Care 2016*. Retrieved May 17, 2020 from <https://digital.nhs.uk/data-and-information/looking-after-information/data-security-and-information-governance/codes-of-practice-for-handling-information-in-health-and-care/records-management-code-of-practice-for-health-and-social-care-2016>
- [54] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.
- [55] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. 2014. IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8767)*, Jürgen Dingel, Wolfram Schulte, Isidro Ramos, Sílvia Abrahão, and Emilio Insfrán (Eds.). Springer, 653–669. https://doi.org/10.1007/978-3-319-11653-2_40
- [56] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedűs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98 (2015), 80–99. <https://doi.org/10.1016/j.scico.2014.01.004>
- [57] Thomas Vogel and Holger Giese. 2010. Adaptation and Abstract Runtime Models. In *SEAMS'10*. ACM, 39–48. <http://dx.doi.org/10.1145/1808984.1808989>
- [58] Thomas Vogel, Andreas Seibel, and Holger Giese. 2010. The Role of Models and Megamodels at Runtime. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2-8, 2010, Reports and Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6627)*, Jürgen Dingel and Arnor Solberg (Eds.). Springer, 224–238. https://doi.org/10.1007/978-3-642-21210-9_22
- [59] Danny Weyns and Radu Calinescu. 2015. Tele Assistance: A Self-Adaptive Service-Based System Exemplar. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, Paola Inverardi and Bradley R. Schmerl (Eds.). IEEE Computer Society, 88–92. <https://doi.org/10.1109/SEAMS.2015.27>