Incremental execution of temporal graph queries over runtime models with history and its applications

Lucas Sakizloglou^{1,2} · Sona Ghahremani¹ · Matthias Barkowsky¹ · Holger Giese¹

Received: 8 March 2021 / Revised: 25 October 2021 / Accepted: 27 October 2021 / Published online: 20 December 2021
© The Author(s) 2021

Abstract

Modern software systems are intricate and operate in highly dynamic environments for which few assumptions can be made at design-time. This setting has sparked an interest in solutions that use a runtime model which reflects the system state and operational context to monitor and adapt the system in reaction to changes during its runtime. Few solutions focus on the evolution of the model over time, i.e., its history, although history is required for monitoring temporal behaviors and may enable more informed decision-making. One reason is that handling the history of a runtime model poses an important technical challenge, as it requires tracing a part of the model over multiple model snapshots in a timely manner. Additionally, the runtime setting calls for memory-efficient measures to store and check these snapshots. Following the common practice of representing a runtime model as a typed attributed graph, we introduce a language which supports the formulation of temporal graph queries, i.e., queries on the ordering and timing in which structural changes in the history of a runtime model occurred. We present a querying scheme for the execution of temporal graph queries over history-aware runtime models. Features such as temporal logic operators in queries, the incremental execution, the option to discard history that is no longer relevant to queries, and the in-memory storage of the model, distinguish our scheme from relevant solutions. By incorporating temporal operators, temporal graph queries can be used for runtime monitoring of temporal logic formulas. Building on this capability, we present an implementation of the scheme that is evaluated for runtime querying, monitoring, and adaptation scenarios from two application domains.

Keywords Runtime models · History-awareness · Historic data · Temporal graph queries · Incremental pattern matching · Runtime monitoring · Self-adaptive systems

1 Introduction

The Model-Driven Engineering (MDE) approach to software development focuses on managing the complexities arising in

Communicated by S. Abrahão, E. Syriani, H. Sahraoui, and J. de Lara.

✉ Lucas Sakizloglou
lucas.sakizloglou@hpi.de

Sona Ghahremani
sona.ghahremani@hpi.de

Matthias Barkowsky
matthias.barkowsky@hpi.de

Holger Giese
holger.giese@hpi.de

¹ Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

² Software System Engineering Group, Brandenburg Technology University, Cottbus, Germany

every stage of the software development cycle [22]. In MDE, models constitute abstractions of complex issues originating in the problem-domain which are systematically transformed into implementation-domain artifacts via computer-based technologies [43]. Modern software systems are intricate and operate in highly dynamic environments for which few assumptions can be made at design-time. This setting has sparked an interest in sophisticated approaches to system monitoring and adaptation at *runtime*, i.e., during the system execution, which aim to mitigate uncertainty. At the same time, these approaches have highlighted the complexities of these activities. Owing to the effectiveness of models at design-time, numerous initiatives surfaced [103] which use models which capture a snapshot of the system constituents as well as their state, i.e., *runtime models* [16,21], to realize monitoring [24,28] and adaptation [41,47,107] solutions.

Thus far the primary concern of these solutions has been the attainment and handling of runtime models which rep-

Fig. 1 Overview of system and INTEMPO interaction

represent the most recent snapshot; the evolution of the model, i.e., its history, has been generally neglected [16,17] although its key role in enabling more informed decision-making during the system lifetime [38,45] and postmortem analysis [14] has been long recognized. Only recently did solutions surface which exploit this potential, e.g., via detection of recurrent behavior patterns and behavior explanation [44], analysis of temporal requirements [84], or inference of probabilistic adaptations based on past interactions [40].

Representing and utilizing history pose significant challenges to solutions which rely on runtime models: representing history requires the representation of a collection of snapshots—rather than the most recent one—through which changes to the model can be tracked; utilizing history relies on the ability to *query* history in an effective and intuitive manner, i.e., on the ordering as well as the (real) time in which changes in the structure of a snapshot occurred. The challenges are only exacerbated in case the model is to be queried *online*, i.e., while the system is running, and the monitoring or adaptation solution depends 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 handle the history of a runtime model: *R1*—an *encoding* of history which represents multiple snapshots as well as enables tracing the occurrence and timing of changes in-between snapshots; *R2*—a *query language* which supports statements on the model structure as well as the ordering and (quantitative) timing in which structural changes occur; *R3*—*fast query execution*, as for monitoring or adaptation scenarios query answers might be necessary for decision-making at runtime; *R4*—a *memory-efficient representation* which may save resources and expedite query executions.

Following the common practice of representing a runtime model as a typed attributed graph, we present a query language and a querying scheme which together fulfill the requirements above. An abstract overview of the system interaction with the scheme, named INTEMPO, is shown in Fig. 1. The language incorporates a temporal logic defined on graphs, thus enabling the formulation of *temporal graph queries*, i.e., graph-based model queries with ordering and timing constraints on the occurrence of graph structures (*R2*). The querying scheme, i.e., a collection of inter-dependent operations, iteratively processes *timed events*, i.e., changes to the system which represent its history, and makes the cor-

responding modifications to a model-based history encoding $\mathcal{R}$ (*R1*). After each modification, INTEMPO executes input queries on $\mathcal{R}$ and returns the *answers* to the system.

Regarding fast query executions (*R3*), we present an operationalization framework which automatically maps a temporal graph logic formula to a network of simple graph sub-queries therefore enabling incremental execution of temporal graph queries. For memory-efficiency (*R4*), we present an optional method which, based on the timing constraints of the formula, derives a window during which model elements are relevant to query executions. Elements outside the window can be *pruned* from the model thus reducing its size while overall returning the same results as un-pruned models.

To demonstrate the effectiveness of INTEMPO we present an implementation based on the Eclipse Modeling Framework [36,101]. The implementation is integrated with a feedback control loop which instruments system adaptation, i.e., an *adaptation engine* [66], and evaluated via simulations based on a case-study of a smart medical system, a real medical guideline, and a combination of real and synthetic event logs. The logs are used for simulations in which query answers from INTEMPO are used while planning adaptations. The case-study is of particular relevance as in healthcare, real-time requirements are key to medical procedures [26] and therefore fast query executions are necessary. The performance of the implementation, i.e., its fulfillment of *R3* and *R4*, is compared to a relevant tool for runtime monitoring as well as a tool from the MDE community. Moreover, we use data generated by the LDBC Social Network Benchmark [75] to test the querying performance of the implementation against larger and more complex graph structures.

This article is an extension of a paper published in MoDELS '20 [94]. Besides improvements on the presentation, structure, and technical explanations, the following contributions are novel with respect to its precursor: (i) the provision of formal arguments on the correctness of answers returned by INTEMPO (ii) the expansion of the concept of discarding history, i.e., definition of projected answer over partial history representations and support for multiple queries (iii) the generalization of the usage of temporal graph queries for monitoring temporal properties against the history of a system—which was treated ad hoc in [94] (iv) an additional comparison to a tool from the MDE community capable of querying history (v) a newly introduced evaluation based on an independent benchmark (vi) an extension of the discussion of related work.

The rest of the paper is organized as follows. Section 2 discusses the case-study and the foundations of INTEMPO. A technical overview of the scheme is presented in Sect. 3. The introduced query language for temporal graph queries is presented in Sect. 4, while Sect. 5 presents the operationalization framework which enables incremental query execution. Section 6 presents a method to discard elements

Fig. 2 Metamodel of SHS (excerpt)

that are not relevant to query executions. Section 7 contains the extensions made to INTEMPO to be suitable for intuitive monitoring of future temporal properties, and Sect. 8 presents the application of the scheme in a self-adaptation scenario. We evaluate the performance of our implementation in Sect. 9, discuss related work in Sect. 10, and conclude the paper as well as discuss future work in Sect. 11. Appendix contains technical preliminaries, proofs of formal arguments, and supplemental information on the evaluation.

2 Prerequisites

This section introduces the case-study from the medical domain, which we use as a running example in the remainder. Moreover, this section presents the foundations of INTEMPO.

2.1 Smart healthcare system

The case-study is based on a service-based simulated Smart Healthcare System (SHS). The SHS is based on smart medical environments [92] 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. Figure 2 depicts the *metamodel* of the SHS (based on the well-known Ecore syntax [36]) which defines valid model instances [19]. The SHS metamodel is based on the exemplar of a service-based medical system in [109] and captures the running system as an instance of the *Architecture* class.

In the SHS, each patient is connected to a sensor. Services are invoked by a main service called *SHSService* to collect measurements from sensors, i.e., *PMonitoringService*, or take medical actions via smart medical devices such as a pump, i.e., *DrugService*. Invocations are triggered by effectors (*Effector*) and invocation results are tracked via

Fig. 3 Rudimentary example of an RTM

monitoring probes (*Probe*) that are attached to *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 *pID* attribute which identifies the patient for whom the *Service* is invoked. Elements in gray are explained later in the article.

2.2 History and time domain

We identify the system behavior with a (possibly infinite) sequence of instantaneous *timed events* which represent observable actions or state changes made by the system or its context at some time point. The system has a clock whose time domain is the set of non-negative real numbers $\mathbb{R}_0^+$. An element of the time domain is called a *time point*.

Intuitively, the *history* of a system with respect to a timed event is the sequence of all observed timed events up to and including said timed event. Technically, the history corresponds to a finite prefix of the behavior consisting of pairs (e^i, τ) with e^i a timed event from a set of possible observations of interest $\mathcal{E}$, $i \in \mathbb{N}$ the position of the event in the index, and $\tau \in \mathbb{T} \mathbb{R}_0^+$ the time point of occurrence. We use the shorthand e_τ when the index position is irrelevant and the shorthand τ_i to denote the time point at position i . Note that, for presentation purposes, we group all changes with the same time point in one event. However, we require that time in the history eventually diverges, i.e., ruling out Zeno behaviors and ensuring that no event groups an infinite amount of changes.¹

For example, for three events e_2, e_4, e_5 , the encompassing history at time point 5 is denoted $\bar{h}_5 := e_2e_4e_5$. When a new event e_7 occurs, the history is incremented by a concatenation to reflect the new history at time point 7, i.e., $\bar{h}_7 = \bar{h}_5 \cdot e_7$.

2.3 Runtime models and history

A (structural) *Runtime Model* (RTM) is a snapshot of the constituents of the modeled system and their state [15,21].

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.

Fig. 4 History $\bar{h}_7^G$ comprising RTMs

See Fig. 3 for an example of a rudimentary RTM (capturing a fragment of the Architecture instance) based on the metamodel in Fig. 2. RTMs are *causally connected* to the modeled system: if the RTM is modified, the system mirrors the modification and vice versa. The causal connection between the system and the RTM may be utilized to alter the system via *model transformation* [49] where *model queries* search for parts of the model that are to be altered via in-place transformations which correspond to desired alterations to the system. By virtue of the causal connection, the alterations are enacted on the modeled system.

A history can be represented by a sequence of RTMs. In this representation, each member is associated with the time point of an event e_τ and mirrors the changes corresponding to e_τ in the system. For instance, represented by RTMs, $\bar{h}_5^G$ corresponds to the sequence $\bar{h}_5^G := G_2G_4G_5$ (see Fig. 4); each RTM 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 $\emptyset$ which is at the start of every such sequence. For spawning an RTM based on an event, we assume that there exists a mapping from the set of events $\mathcal{E}$ to corresponding model modifications. In this case, according to the mapping, event e_7 corresponds to the association of s with the newly added pm_2 and the disassociation of s from the deleted d_1 —when the event occurs, owing to causal connection, it is mirrored in the RTM. To include the latest change, the history representation $\bar{h}_5^G$ is extended by the RTM G_7 , i.e., $\bar{h}_7^G = \bar{h}_5^G \cdot G_7$ —illustrated in Fig. 4.

Fig. 5 RTM^H instances $H_{[5]}$ and $H_{[7]}$

A Runtime Model with History (RTM^H) [95] is an enhanced RTM that simultaneously provides two views on the modeled system: a view of the *current* system state which corresponds to a conventional causally connected RTM; and a compact view of the *history*. The view on history is afforded by each entity being equipped with a *creation* timestamp and a *deletion* timestamp, abbreviated *cts* and *dts*, respectively. For an example, see Fig. 2 where all entities inherit from the *MonitorableEntity*.

The *cts* and *dts* capture the time points of creation and deletion of an entity, respectively, by an event. Upon the occurrence of an event, similarly to an event yielding a new system state, the corresponding entity creation (deletion) in the RTM^H yields a new instance of the RTM^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 ∞ . When an entity is deleted in the modeled system, the respective entity is *not* deleted in the RTM^H. Rather, its *dts* is updated to the time point of the event that induced the deletion.

We assume that connectors in an RTM^H exist for as long as both their end-points exist. Moreover, attribute values in an RTM^H are set when the entities are created and, once set, remain unchanged. If changes of attribute values (or connectors) in the modeled system are of interest, an RTM^H can track such changes by appropriate modeling decisions, e.g., by encoding these elements as entities in the metamodel—as shown in [70]. Each value change would then lead to a creation of a new entity in the RTM^H, where the duration of the

value would be captured by a *cts* and a *dts*. We demonstrate such an encoding in the evaluation in Sect. 9.4.

By retaining all entities as well as information on their creation and deletion time points, an RTM^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 so as to be able to track the evolution of entities across the sequence, an RTM^H stores only a single instance of each model entity. Hence, it affords a compact and, therefore, more efficient representation of history. Similarly to an RTM, an instance of the RTM^H is always associated with the time point of the latest event. For example, the representation of $\bar{h}_7^G$ by an RTM^H yields a single model, $H_{[7]}$, that contains the same information as $\bar{h}_7^G$. $H_{[7]}$ is illustrated in Fig. 5. We note that, in an RTM^H , causal connection only applies to the latest snapshot.

Technically, an RTM^H is obtained by an iterative *coalescence* of a new event and an RTM^H into a new RTM^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 set to the time point of the event. We denote this coalescence by $\odot$. For example, when $\bar{h}_5^G$ is incremented by the event e_7 which corresponds to the creation of pm_2 and the deletion of d_1 , this event yields a new RTM^H $H_{[7]} = H_{[5]} \odot e_7$, illustrated in Fig. 5—attributes updated from the previous RTM^H are accentuated.

2.4 Graph-based RTMs, transformation, and queries

An RTM is often encoded as a *typed, attributed graph* [37] where entities are modeled as vertices, connectors between entities as edges, and information about entities as vertex attributes [106]. A typed, attributed graph (henceforth, simply referred to as a graph) is *typed* over a *type graph* which defines types of vertices, edges, and attributes—similarly to the relationship between a metamodel and a model. In this context, the metamodel in Fig. 2 may be seen as an informal representation of the type graph of the SHS. The RTM^H is analogously based on *graphs with history* [100].

Attributes are associated with a data type, i.e., a character string, an integer, a real number, or a Boolean. Graphs contain a set of *assignments* A which assign data-type-compatible values to attributes, e.g., $\text{pm}_1.\text{PID} = 1$ in Fig. 3. Given that attribute values in an RTM^H are set upon the creation of an entity and remain unchanged (see Sect. 2.3), assignments in A are fixed. Formally, the set of assignments A may have various representations, e.g., distinguished data vertices [37] or an attribute constraint over sorted variables [99].

Encoding an RTM as a graph allows for the realization of model transformation via established formalisms, such as *typed, attributed graph transformation* [37] where *graph transformation rules* are used to search for a part of the model which is transformed in place [49].

In short, let G be a graph (in this context, encoding an RTM) typed over a type graph T and ρ a graph transformation rule. The rule ρ is characterized by a left-hand side (*LHS*) and a right-hand side (*RHS*) graph, also typed over T , which define the pre-condition and postcondition of an application of ρ , respectively. Intuitively, the execution of ρ searches for *LHS* in G and transforms it according to *RHS*. Moreover, the *LHS* graph may be extended with a Boolean expression γ (additionally to the set of assignments A) over the values of attributes in A . We refer to this *LHS* graph extended by γ as a (graph) *pattern* and to G as a *host graph*.

The *LHS* of a rule characterizes a (graph) *query*, which is the equivalent graph-based notion of a model query. The execution of a graph query over a given host graph, also called (graph) *pattern matching*, amounts to finding *matches*, i.e., occurrences of the query pattern in the host graph, whose attribute values satisfy the assignments and attribute constraint in A and γ of the pattern, respectively.

Formally, a match is a mapping from the pattern in the query to the host graph which preserves structure and type—also called a *morphism*. In the following, we use the two terms interchangeably. The query *answer set* is a set strictly containing all matches for a query in G . The transformation of the rule, specified in the *RHS*, is performed only when a match for the query, i.e., the *LHS*, has been found. The match identifies a part of G where the transformation should occur.

Queries with Complex Patterns In certain cases, simple patterns are not sufficient as a language for defining more complex *application conditions* of rules, for instance if the existence of certain model elements should be prohibited. In those cases, an *LHS*, i.e., graph query, is enhanced with an application condition *ac* which every match m should satisfy. In the following, a query θ is characterized by a pattern n and an application condition *ac*, and denoted $\theta := (n, ac)$.

The language of *Nested Graph Conditions* (NGCs) [55] can formulate *ac* that are as expressive as first-order logic on graphs [27] as shown in [55,88] and constitute, as such, a natural formal foundation for pattern-based queries. NGCs support standard first-order logic operators. The syntax of an NGC ϕ is given by the grammar:

$$\phi ::= \text{true} \mid \neg\phi \mid \phi \wedge \phi \mid \exists(n \hookrightarrow \hat{n}, \phi)$$

where $n, \hat{n}$ are patterns. The existential quantifier features a morphism (denoted by a hooked arrow) from n to $\hat{n}$ which relates, i.e., *binds*, elements in outer conditions (n) to inner (nested) conditions ($\hat{n}$) and is therefore also called a *binding*.

Fig. 6 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 object in the host graph

Let $\mathcal{L}$ be the language for queries with ac based on NGC and $\theta := (n, \phi)$ with $\theta \in \mathcal{L}$, i.e., ϕ is an NGC. The answer set $\mathcal{A}$ for θ over a host graph G contains all matches m in G for the pattern n that satisfy ϕ . We also denote $\mathcal{A}$ by $\mathcal{A}(G)$ when θ is clear from the context. 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 is 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. In the remainder, we abbreviate $\neg(\neg\phi \wedge \neg\hat{\phi})$ by $\phi \vee \hat{\phi}$, $\exists(n \hookrightarrow \hat{n}, \phi)$ by $\exists(\hat{n}, \phi)$, and $\exists(n, \text{true})$ by $\exists n$.

As an example, assume the following hypothetical requirement which draws from *operation sequence compliance*, i.e., the order of service invocations, in an SHS [see 109]: “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 Fig. 2, 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 Fig. 6, 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 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 $\neg\exists n_{1.1} \wedge \exists n_{1.2}$. Nesting implies that the 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 $\mathcal{A}(G_5)$ —from Fig. 4—for θ_1 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.

Query Operationalization A graph query is a declarative means to express a structure of interest which should satisfy a given condition. The query itself does not specify instructions on how to execute the query, i.e., its *operationalization*. For the operationalization of queries, we build on a formal framework we have previously presented which supports queries in $\mathcal{L}$. The framework in question decomposes a query with an arbitrarily complex NGC as ac into a suitable ordering of simple, pattern-based sub-queries called a *Generalized Discrimination Network* (GDN) [18].

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 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 realized 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 executed bottom-up: the execution starts with leaves and proceeds upward in the network. The terminal node computes the $\mathcal{A}$ of the query.

In [18], 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 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 transformation rules of a GDN, henceforth called *marking rules*, 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 rules 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.

The GDN for θ_1 from earlier is shown in Fig. 7, 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 $N_{1.1}$ searching for the pattern $n_{1.1}$, the node $N_{1.2}$ searching for $n_{1.2}$, and the topmost node N_1 searching for n_1 . Node N_1 computes its matches by matching its pattern and checking whether both of its dependencies are satisfied (the conjunction in ϕ_1). The negative dependency which captures the negation in ϕ_1 (drawn with a dashed line) is satisfied when a match for N_1 cannot be extended by a match for $N_{1.1}$. All nodes are realized by marking rules whose *LHS* matches a pattern and whose *RHS* creates marking nodes and edges that

Fig. 7 GDN and marking rules for θ_1 —based on [18]

mark the matches of the *LHS*. The rules for nodes $N_{1.1}$, $N_{1.2}$ are shown in Fig. 7 (within rectangles), where (i) marking nodes are illustrated by circles and (ii) the marking nodes and edges added by a rule are dashed and annotated with “++”. For presentation purposes, the illustrations of rules thus contain both their *LHS* and *RHS*.

Incremental Execution Optimization techniques such as *local search* can be employed to reduce the pattern matching effort of GDN nodes [see 3]. 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 as well as local search, a GDN is amenable to *incremental execution*. Changes in G are propagated through the network, whose nodes only recompute their results if the change concerns them or one of their dependencies. 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. Therefore, we say that the query is also executed incrementally as its $\mathcal{A}$ is updated by each GDN execution.

2.5 Metric temporal graph logic

Metric Temporal Graph Logic (MTGL) [50] enables the formulation and checking of *temporal requirements* on patterns, i.e., requirements on the evolution of patterns over time.

MTGL builds on NGCs and *Metric Temporal Logic* [69] to enable the definition of *Metric Temporal Graph Conditions* (MTGCs) on patterns. Additionally to the NGC operators, MTGCs support *metric*, i.e., interval-based, *temporal* operators: the *until* (U_I , with I an interval in $\mathbb{R}_0^+$) and its dual *since* (S_I). The syntax of an MTGC ψ is given by:

$$\psi ::= \text{true} \mid \neg\psi \mid \psi \wedge \psi \mid \exists(n, \psi) \mid \psi U_I \psi \mid \psi S_I \psi$$

with n a pattern. The operators *eventually* ($\Diamond_I$) and *once* ($\blacklozenge_I$) are abbreviations of *until* and *since*: $\Diamond_I \psi = \text{true} U_I \psi$ and $\blacklozenge_I \psi = \text{true} S_I \psi$. We abbreviate *exists* similarly to NGCs.

MTGL reasons over sequences of graphs. Intuitively, this is motivated by the logic expressing requirements on the evolution of a pattern over time, i.e., over consecutive graphs. However MTGCs can also be equivalently checked over a graph with history [50], which here corresponds to an RTM^H . Following is the intuition behind satisfaction for the MTGL operators $\exists$, U_I , S_I .

Let ψ be the MTGC $\exists(n, \hat{\psi})$. A match m for n in an $\text{RTM}^H_{[T]}$ satisfies ψ at time point $\tau \in \mathbb{T}$ if $\max_{e \in E} e.cts \leq \tau < \min_{e \in E} e.dts$, with E the elements of m , and matches for $\hat{\psi}$ are compatible with m and satisfy $\hat{\psi}$. Given two MTGCs $\psi, \hat{\psi}$, the MTGC $\psi U_I \hat{\psi}$ is satisfied at a time point τ when there is a time point τ' with $\tau' - \tau \in I$, where $\hat{\psi}$ is satisfied, and at least for all $\tau'' \in [\tau, \tau')$, ψ is satisfied. The intuition is reversed for $\psi S_I \hat{\psi}$ which is satisfied at a time point τ , when there is a τ' with $\tau - \tau' \in I$, where $\hat{\psi}$ is satisfied, and at least for all $\tau'' \in (\tau', \tau]$, ψ is satisfied.

3 The INTEMPO scheme

An overview of the functionality of INTEMPO is provided in Sect. 1 and illustrated in Fig. 1. Building on the technical prerequisites in Sect. 2, this section presents INTEMPO in further technical detail—see Fig. 8 for a graphical reference—and shows how the requirements for a history encoding ($R1$), a query language ($R2$), fast answers ($R3$), and memory-efficiency ($R4$) defined in Sect. 1 are fulfilled.

Design-time 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 vertex (ii) a *mapping* of events to model modifications, e.g., additions or deletions of nodes.

Runtime To represent the history of a system, i.e., for $R1$, INTEMPO relies on an RTM^H (see Sect. 2.3) which, by featuring creation and deletion timestamps, captures temporal information on multiple past versions of an RTM into a single, consolidated representation.

For $R2$, we introduce a novel language (see Sect. 4) which allows for the specification of *temporal graph queries*, i.e., queries on the ordering and timing in which patterns are added or deleted in the RTM^H . Compared to the queries with NGCs presented in Sect. 2.4, temporal graph queries use MTGCs (see Sect. 2.5) for the formulation of *ac*, i.e., they incorporate (past and future) temporal operators with timing constraints, and, moreover, compute the interval for which matches for a temporal graph query satisfy the query’s *ac*.

INTEMPO is invoked by the system and provided with a sequence of changes encapsulated in (timed) *events*. The

Fig. 8 Overview of INTEMPO Operations

operation of INTEMPO at runtime is *reactive*. First, upon the occurrence of an event, the system invokes INTEMPO. Then, INTEMPO consults the event mapping, makes the corresponding modifications to the RTM^H , and sets the values of the creation and deletion timestamps based on the time point of the event. Finally, the scheme executes the input queries for each modification and returns the answers to the system.

For systems with lengthy histories or a high rate of incoming events, the re-computation of matches from scratch upon every event would quickly lead to slow query executions, arguably rendering a runtime solution impractical. Instead, for the execution of temporal graph queries over an RTM^H , INTEMPO uses a novel operationalization framework which supports incremental execution of temporal graph queries (Sect. 5). This feature allows for fast query executions (*R3*) and also gives INTEMPO its name: *Incremental execution of Temporal graph queries*. Aiming for fast executions and memory-efficiency, INTEMPO supports an optional history representation which contains only elements that are relevant to query executions (Sect. 6). Queries over this constrained RTM^H may yield faster executions while affording increased memory-efficiency.

Operations INTEMPO consists of two core operations: *Operationalization* and *Execution*. *Maintenance* constitutes an optional extension which, if enabled, is performed after Execution. We outline each operation below.

Operationalization: This operation constructs a *temporal GDN* for each of the input temporal graph queries. The operation extends the GDN construction presented in Sect. 2.4 by introducing concepts for handling structural matches whose validity is based on the *cts* and *dts* of their elements as well as for evaluating the *until* and *since* operators from MTGL. This operation is only performed once per query (at the beginning) and if the set of input queries has changed between invocations of INTEMPO.

Execution: The operation executes the temporal GDN(s) over the RTM^H . A temporal GDN processes the modifications made to the RTM^H and, for modifications that are relevant to the queries, executes the affected sub-queries. The sub-queries search for matches of patterns while taking into

account timing constraints on the occurrence of patterns. By virtue of the incremental execution of the GDN, matches are updated incrementally after each event.

Maintenance: The operation relies on the computation of a *time window* (during Operationalization) which is based on timing constraints of the temporal operators in input queries. The operation uses the window to decide when deleted elements are not going to be involved in future query executions and can be thus *pruned* from the RTM^H .

4 Language for temporal graph queries

Temporal requirements are always present in medical guidelines and their satisfaction is key to the successful completion of procedures [26]. Adding a temporal dimension to the exemplary requirement from Sect. 2.4 makes it similar to compliance checking of medical procedures which may track time between triage and admission [78], here represented by the invocation of a sensor service (n_1) and a drug service ($n_{1,2}$), respectively: “When a sensor 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 in a query, this instruction includes temporal requirements on the evolution of graph structures: “find all matches for n_1 such that, for 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 time units 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 pm and by time unit we refer to the unit of measurement with which the system tracks time—here, assumed to be a second.

The language $\mathcal{L}$ introduced in Sect. 2.4, which employs NGCs for the definition of *ac*, does not inherently support temporal requirements on patterns. MTGCs, introduced in Sect. 2.5, build on NGCs and allow for the specification of a desired ordering and a timing constraint on the evolution of graph structures—thereby supporting temporal requirements. Moreover, in an encoding where elements have lifespans, a structural occurrence of a pattern consisting of such elements has to be accompanied by information on when and for how long a match exists. This requirement is emphasized for application scenarios where a decision may depend on timing, e.g., in runtime adaptation.

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, we introduce the query language $\mathcal{L}_T$ which enables the formulation of *temporal graph queries* over an RTM^H . A temporal graph query $\zeta \in \mathcal{L}_T$ is characterized similarly to graph queries with NGCs in $\mathcal{L}$, i.e., $\zeta := (n, ac)$. However, in

contrast to $\mathcal{L}$, temporal graph queries feature *ac* based on MTGCs thereby supporting temporal requirements. In $\mathcal{L}_T$, the exemplary query above is captured by $\zeta_1 := (n_1, \psi_1)$ where ψ_1 is an MTGC defined as $\neg \exists n_{1.1} \cup_{[0,60]} \exists n_{1.2}$. Recall that elements common to n_1 and the patterns $n_{1.1}, n_{1.2}$ are bound—see Sect. 2.4.

Compared to $\mathcal{A}$, an answer set $\mathcal{T}$ for a query $\zeta \in \mathcal{L}_T$ is extended with a temporal dimension: matches in $\mathcal{T}$ are paired with a *temporal validity*, i.e., the set of time points for which (i) matched elements co-exist in the RTM^H and satisfy the attribute constraint, called *lifespan* of the match (ii) the match satisfies the *ac* of ζ , called *satisfaction span*. We elaborate on the lifespan, the satisfaction span, and the temporal validity below.

4.1 Lifespan of a match

Vertices of an RTM^H have attributes which capture their creation (*cts*) and deletion (*dts*) timestamps. For an element ϵ , we define its *lifespan* as the non-empty non-negative interval $[\epsilon.cts, \epsilon.dts)$. The intuition behind the lifespan of an element being right-open² is that if an element has been deleted at a time point τ this means the element has existed until a time point that approaches but is not equal to τ . A match 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 elements E overlap:

$$\lambda^m = \bigcap_{\epsilon \in E} [\epsilon.cts, \epsilon.dts) \quad (1)$$

Attribute values of matched elements do not change (see Sect. 2.4) and, hence, cannot affect the lifespan computation. Element 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 in ζ is the empty graph $\emptyset$, an (empty) match m is always found with $\lambda^m = \mathbb{R}_0^+$.

4.2 Satisfaction span and temporal validity

We call the set of time points for which an MTGC is satisfied its *satisfaction span*, denoted by $\mathcal{Y}$. In the context of a query $(n, \psi) \in \mathcal{L}_T$ or a nested condition with n as enclosing pattern, a satisfaction span related to a match m for n 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 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 $I \in \mathcal{J}$ with $\mathcal{F}$ and $\mathcal{J}$ 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 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 . The *satisfaction computation* $\mathcal{Z}(m, \psi)$ is recursively defined as follows.

$$\mathcal{Z}(m, \text{true}) = \mathbb{R} \quad (2)$$

$$\mathcal{Z}(m, \neg \chi) = \mathbb{R} \setminus \mathcal{Z}(m, \chi) \quad (3)$$

$$\mathcal{Z}(m, \chi \wedge \omega) = \mathcal{Z}(m, \chi) \cap \mathcal{Z}(m, \omega) \quad (4)$$

$$\mathcal{Z}(m, \exists(\hat{n}, \chi)) = \bigcup_{\hat{m} \in \hat{M}} \lambda^{\hat{m}} \cap \mathcal{Z}(\hat{m}, \chi) \quad (5)$$

$$\mathcal{Z}(m, \chi \cup I \omega) = \begin{cases} \bigcup_{i \in \mathcal{Z}(m, \omega), j \in J_i} j \cap ((j^+ \cap i) \ominus I) & \text{if } 0 \notin I \\ \bigcup_{i \in \mathcal{Z}(m, \omega)} i \cup \bigcup_{j \in J_i} j \cap ((j^+ \cap i) \ominus I) & \text{if } 0 \in I \end{cases} \quad (6)$$

$$\mathcal{Z}(m, \chi S_I \omega) = \begin{cases} \bigcup_{i \in \mathcal{Z}(m, \omega), j \in J_i} j \cap ((^+ j \cap i) \oplus I) & \text{if } 0 \notin I \\ \bigcup_{i \in \mathcal{Z}(m, \omega)} i \cup \bigcup_{j \in J_i} j \cap ((^+ j \cap i) \oplus I) & \text{if } 0 \in I \end{cases} \quad (7)$$

with:

- $\hat{M}$ a set containing only matches that are compatible with the (enclosing) match m —see Sect. 2.4
- 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 k left-closed, and k^+ defined symmetrically
- $k \oplus l = [\ell(k) + \ell(l), r(k) + r(l)]$, $k \ominus l = [\ell(k) - r(l), r(k) - \ell(l)]$ for $k, l \in \mathcal{J}$ 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 (sub-)query 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., $\ell(I) \neq 0$, the satisfaction includes

² An interval is right-open when its right end-point is excluded from the interval. See Appendix A for a comprehensive definition of intervals.

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 span until the very last time point, i.e., *adjacent*, before 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 j' is empty, i.e., does not exist, and in that case *until* is satisfied by 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 1 *Given a match m and an MTGC ψ , the satisfaction span $\mathcal{Y}$ of m for ψ is given by the satisfaction computation $\mathcal{Z}$ of m for ψ , that is, $\mathcal{Y}(m, \psi) = \mathcal{Z}(m, \psi)$*

Proof (sketch) By structural induction over ψ . For every equation, inclusion is shown in both directions. See Sect. B.1 for the proof. $\square$

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 2 (*temporal validity $\mathcal{V}$*) Let n be a pattern, ψ an MTGC, and $H_{[\tau]}$ an RTM^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 to take elements from the match m into account but it is not bound by it. The temporal validity $\mathcal{V}(m, \psi)$ binds $\mathcal{Z}$ 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 $(n, \Diamond_{[0,5]}\exists \hat{n})$. For a match m for n with $\lambda^m = [3, 9]$, the satisfaction span $\mathcal{Z}(m, \Diamond_{[0,5]}\exists \hat{n})$ for a match $\hat{m}$ for $\hat{n}$ with $\lambda^{\hat{m}} = [3, 6]$ is satisfied for $[-2, 6]$ —according to Eq. 6. However, $\mathcal{V}(m, \Diamond_{[0,5]}\exists \hat{n})$ 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 1), whereas $\mathcal{V} \subseteq \mathbb{R}_0^+$ —since $\mathcal{V}$ is produced by an intersection with λ^m . It also holds that $\mathcal{V}(m, \psi) \subseteq \mathcal{Z}(m, \psi)$.

4.3 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 vertices and, thus, all matches

have a lifespan. Moreover, application conditions of queries in $\mathcal{L}_T$ are formulas of an interval-based temporal logic which decides the truth of a formula for every time point in $\mathbb{R}$. Therefore, the answer set of a query in $\mathcal{L}_T$ contains matches, i.e., structural occurrences of a pattern, associated with a temporal validity, i.e., the time points for which the match exists *and* satisfies its application condition.

Definition 3 (*answer set $\mathcal{T}$*) Given a pattern n , an MTGC ψ , and an RTM^H $H_{[\tau]}$, the *answer set* $\mathcal{T}$ for a query $\zeta := (n, \psi)$ over $H_{[\tau]}$ is given by:

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

Recall that $\mathcal{V}(m, \psi)$ may be 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 and, moreover, to define a restricted answer set for queries over a constrained RTM^H.

5 Operationalization of temporal graph queries

This section presents an operationalization framework that enables the incremental execution of a query $\zeta \in \mathcal{L}_T$. In INTEMPO, the activities described below are performed by Operationalization and Execution. Given ζ , Operationalization constructs an enhanced GDN which is capable of interval operations. Execution executes this GDN to find and update matches for ζ in an RTM^H.

In the following, we refer to the framework in [18] (see Sect. 2.4) as *base approach*. The base approach considers graph queries in $\mathcal{L}$, i.e., where application conditions are formulated as NGCs, and thus does not support the temporal operators *until* and *since* and, in general, the temporal reasoning required by MTGL. Building on the base approach, we present our extensions to which we collectively refer as *temporal approach* or temporal GDN. The temporal GDN supports the formulation of an *ac* in MTGL and therefore enables incremental execution of queries in $\mathcal{L}_T$.

5.1 Marking rules of temporal GDN

Regarding marking rules, the main difference between the base approach and the temporal approach is that the *RHS* of rules of the temporal approach create a marking node that captures the *duration* of the match being marked. To this end, the type of marking nodes in the type graph is equipped with an attribute d of type interval set.

In detail, first, we adjust the concept of a regular marking rule of the base approach such that we obtain two variants: one where the *RHS* creates a marking node with a duration

Fig. 9 Temporal GDN: network and marking rules for ζ_1 —the computation of the duration of the created marking node by network nodes $N_1, N_{1,1}, N_{1,2}$ is based on Definition 2, by $\alpha_{1,1}, \alpha_{1,2}$ on Eq. 8, and by U on Eq. 6

that coincides with the temporal validity $\mathcal{V}$ of the match being marked; and another where the *RHS* creates a marking node with a duration that coincides with the satisfaction span $\mathcal{Z}$ of the match being marked. The two rules are denoted by $\mathcal{V}$ MR and $\mathcal{Z}$ MR, respectively. Moreover, we introduce a new type of marking rule, denoted by α MR, which allows for matching a varying number of nodes, a feature which is not supported by the base approach.

The temporal GDN incrementally updates in each execution step the set of all matches and potentially re-computes the duration of their marking nodes. Once created, marking nodes, together with the vertices they mark, remain in the RTM^H in subsequent steps. We elaborate on each rule below.

The $\mathcal{V}$ Marking Rule A $\mathcal{V}$ MR sets the d of a created marking node to the temporal validity of a match m , given by Definition 2—thus, the rule is intended for *exists* operators. The satisfaction span is represented by the duration of marking nodes of the dependencies of the rule. Since matched elements may be marking nodes, the computation of the lifespan of a match is slightly adjusted. For a matched element ϵ , its lifespan is given by $\epsilon.d$ if ϵ is a marking node, and by $\{\epsilon.cts, \epsilon.dts\}$ otherwise..

Figure 9 contains an example of a $\mathcal{V}$ MR: the node $N_{1,1}$ for $\exists n_{1,1}$ from $\psi_1 := \neg \exists n_{1,1} U_{[0,60]} \exists n_{1,2}$. Marking nodes are illustrated by a split circle where the bottom compartment contains the duration attribute.

Contrary to rules in the base approach, a $\mathcal{V}$ MR includes marking nodes of dependencies in the pattern of the query rather than in the *ac*. This is necessary as these marking nodes are included in computations of the duration of the marking node created by the *RHS*.

The $\mathcal{Z}$ Marking Rule The $\mathcal{Z}$ MR is intended for the operators in Definition 1 which pass their match on to enclosing MTGCs and do not include matched elements in their computations, i.e., the operators negation, conjunction, *until*, and *since*. Therefore, the *LHS* of the $\mathcal{Z}$ MR contains all elements of its closest enclosing $\mathcal{V}$ MR. For example, the U node and the negation node in Fig. 9 contain all elements in n_1 . The $\mathcal{Z}$ MR's computation of d excludes these elements and only considers the satisfaction span of its dependencies, represented by the duration of marking nodes. The computation by the *RHS* depends on the operator—see Definition 1.

The α Marking Rule In Eq. 5, the computation of the satisfaction span for *exists* relies on the union of all lifespans of matches that are compatible with the enclosed pattern $\hat{n}$. In order to compute this union, the corresponding node in the temporal GDN is required to keep track of all matches for $\hat{n}$. The number of these matches might vary in every GDN execution, a property which is not covered by conventional graph transformation rules—see Sect. 2.4.

Therefore, the node created for the *exists* operator is an *amalgamated marking rule* (α MR). Such rules stem from amalgamated graph transformation [20], where an arbitrary number of parallel transformations are amalgamated, i.e., merged, into a single rule execution in one transformation step. Thus, an α MR enables a node to be associated with a varying number of graph elements so that it can compute the union of their duration.

The *LHS* of α MR contains a *kernel* of graph elements that are bound by the enclosing operator—in ψ_1 , that would be the *s*- and a *multi-rule* which matches an arbitrary number of instances of a certain marking node type. An α MR thus *groups* the marking nodes matched by the multi-rule by matches of the kernel and *aggregates* the marking nodes' duration. Hence, the α MR corresponds to a GDN node with a single dependency to the node that creates the marking nodes which the α MR groups.

The *RHS* of an α MR creates an α marking node which is connected to the marking nodes of its dependency (matched by the multi-rule) and the elements of the kernel marked by those marking nodes. Only a single α marking node is created, regardless of the number of matches for the dependency. The duration of the α marking node is the union of the duration of the marking nodes E^M matched by the multi-rule:

$$d^\alpha = \bigcup_{\epsilon \in E^M} d^\epsilon \quad (8)$$

See Fig. 9 for an example of an α MR, the node named $\alpha_{1,1}$, which groups matches of its dependency, the node $N_{1,1}$.

5.2 Temporal GDN construction

In the previous section, we were concerned with types of GDN rules and the types of marking nodes they created. This

section focuses on *GDN nodes* which form the components of the network and represent executions of GDN rules.

In INTEMPO, the construction of a temporal GDN from a query in $\mathcal{L}_T$ is performed by Operationalization—see Fig. 8. Compared to the base approach, the construction features two extensions: first, we translate the operators conjunction, negation, *until*, and *since* into GDN nodes corresponding to $\mathcal{ZMR}$; second, we translate the *exists* operator into a GDN node corresponding to a $\mathcal{VMR}$, a GDN node corresponding to an αMR which is intended for grouping the matches of the node for the $\mathcal{VMR}$, and a positive dependency from the node for the αMR to the node for the $\mathcal{VMR}$.

The construction is performed in two steps: (i) a traversal of the syntax tree of the MTGC of the query where operators are replaced by GDN nodes (ii) a traversal of the network created by the previous step to create the dependencies between nodes and set the patterns of $\mathcal{ZMR}$ nodes.

See Fig. 9 for the temporal GDN of ζ_1 , where novel nodes, i.e., created by our extensions to the base approach, are in light gray. The existential conditions in ψ_1 , i.e., $\exists n_{1,1}$ and $\exists n_{1,2}$, lead to the creation of nodes $N_{1,1}$ and $N_{1,1}$, respectively. These nodes are both dependencies of their respective α nodes. The pattern $n_{1,1}$ is enclosed by a negation which leads to the creation of negation node (“ $\neg$ ”) and a negative dependency between the negation node and $\alpha_{1,1}$. The U node is dependent on the negation node and $\alpha_{1,2}$. The node N_1 , created for the pattern n_1 of ζ_1 , is dependent on the U node. For more complex constructions, the instructions in [18] apply.

5.3 Temporal GDN execution

In INTEMPO, the execution of a temporal GDN is performed by Execution (see Fig. 8). The execution is recursive: given an input GDN node, the node’s dependencies are executed before the input node. Hence, although the execution starts at the root of the GDN, network leaves, which have no dependencies, are executed first.

An execution of the temporal GDN is performed after each modification to the RTM^H and nodes start the pattern matching effort in the surrounding of elements affected by the modification—see local search in Sect. 2.4. The pattern matching for a node is skipped if the modification does not concern the node or one of its dependencies.

We demonstrate an execution of a temporal GDN via an example based on the network constructed for query ζ_1 (see Fig. 9) and two RTM^H instances: $H_{[5]}$ and $H_{[7]}$ in Fig. 5.

Execution over $H_{[5]}$ Each item in the following lists marks the execution of a GDN node. Each GDN node execution leads to the creation of marking nodes whose duration depends on the type of the GDN node—the type of each GDN node is shown between braces next to the node name. The execution traverses the network and starts with node $N_{1,1}$. It

continues upward with nodes whose dependencies have been executed.

$N_{1,1}$ ($\mathcal{VMR}$) No matches for $n_{1,1}$ are found.

$\alpha_{1,1}$ (αMR) The vertices s and pm_1 are found but no marking nodes from its dependency $N_{1,1}$. The marking node created by αMR groups the duration of its dependency based on Eq. 8. Here, an empty duration is found, hence the empty interval is stored into its attribute $d_{1,1}^\alpha$.

$\neg$ ($\mathcal{ZMR}$) The *LHS* of the rule is the same as n_1 . The node finds one match (vertices s and pm_1) and one marking node for its dependency, $\alpha_{1,1}$. The duration d^ν of the node is computed according to Eq. 3: $\mathbb{R} \setminus d_{1,1}^\alpha = \mathbb{R} \setminus \emptyset = \mathbb{R}$.

The negation node is one of the two dependencies of node U . Before proceeding with node U , the execution requires that the other dependency is executed as well, hence, via recursion, it proceeds with node $N_{1,2}$:

$N_{1,2}$ ($\mathcal{VMR}$) One match is found, containing s , pm_1 , d_1 , and the edges among them. One marking node is created with duration $[5, \infty)$, which coincides with the temporal validity of the match computed according to Definition 2.

$\alpha_{1,2}$ (αMR) One match for s and pm_1 is found as well as a marking node of type $N_{1,2}$ —the one created for node $N_{1,1}$. One marking node is created whose duration aggregates the lifespans of its dependencies. Computed based on Eq. 8, the lifespan $d_{1,2}^\alpha$ is equal to $[5, \infty)$.

Since both of its dependencies have been executed, the execution can now proceed with node U :

U ($\mathcal{ZMR}$) The *LHS* of the rule is the same as n_1 . One match for s and pm_1 is found and two marking nodes: one for the left operand, i.e., the negation node, and one for the right operand, i.e., $\alpha_{1,2}$. For each interval in the duration of the marking node $d_{1,2}^\alpha$ it is checked whether it is adjacent to or overlapping with any of the intervals in the duration of the negation node. The interval $\mathbb{R}$ from d^ν and the interval $[5, \infty)$ from $d_{1,2}^\alpha$ are indeed overlapping. Therefore, the duration d_U of the marking node created by $\mathcal{ZMR}$ is computed according Eq. 6 which returns $[-55, \infty)$.

N_1 ($\mathcal{VMR}$) An s and a pm_1 are matched. Their lifespan is $[4, \infty)$. One marking node with duration d_U exists; effectively, this duration is the satisfaction span of the *ac* of the query. The temporal validity of the match is the intersection of the lifespan of the match with the duration, i.e., $[4, \infty) \cap [-55, \infty) = [4, \infty)$.

The executed query returns a $\mathcal{T}$ which contains a match for n_1 associated with the interval $[4, \infty)$ during which, besides

being structurally present in the graph, the match satisfies the temporal requirements expressed by ψ_1 .

Execution over $H_{[7]}$ The event e_7 yields a new RTM^H instance, i.e., $H_{[7]}$ —see Fig. 5. The event corresponds to the deletion of d_1 and the creation of p_{m_2} as well as an edge between p_{m_2} and s . INTEMPO is invoked and Execution executes the temporal GDN:

$N_{1.1}$ ($\forall\text{MR}$) No match for $n_{1.1}$ is found.

$\alpha_{1.1}$ (αMR) A new match for the kernel exists (the one involving p_{m_2}) whose duration is also the empty interval. The match found in the previous execution remains unchanged.

$\neg$ (ΣMR) Recall that the node's *LHS* is the same as n_1 . A new match is found for s and p_{m_2} whose duration is computed to be $\mathbb{R}$, similarly to the previous execution over $H_{[5]}$. The match found in the previous execution remains unchanged.

$N_{1.2}$ ($\forall\text{MR}$) No new matches are found, however, the duration of the match already stored is updated to $[5, 7)$.

$\alpha_{1.2}$ (αMR) A new match with s and p_{m_2} is found for the kernel with an empty duration as no corresponding marking node exists. The duration of the match found in the previous execution, i.e., $H_{[5]}$, is re-computed to $[5, 7)$ to reflect the change in the duration of its dependent marking node in the previous phase.

U (ΣMR) A new match is found by its *LHS*. This match is associated, via the matches stored by the right operand for the until, with an empty interval. The computation of the duration for this match is similarly an empty interval. The duration of one of the marking nodes associated with the match found in the previous execution has been updated: this triggers an update of the duration stored by the node for the match in question. The result of the re-computation is $[-55, 7)$.

N_1 ($\forall\text{MR}$) A new match is found. The satisfaction span for this match is an empty interval and, hence, its temporal validity is also empty. The temporal validity of the previously found match for n_1 is re-computed to $[4, 7)$.

INTEMPO returns a $\mathcal{T}$ which has been incrementally computed and contains one match for n_1 —the one involving p_{m_1} . The match involving p_{m_2} , although structurally present, does not satisfy the MTGC in ζ_1 —the match has an empty temporal validity and is hence excluded from $\mathcal{T}$. The temporal validity of the match involving p_{m_1} has been updated to reflect the change to d_1 by e_7 .

6 Constraining memory consumption of an RTM^H

As noted previously, an RTM^H maintains two views on the state of the modeled system. Similar to traditional RTMs, one view is of the current system state, as the RTM^H contains all system 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) featuring vertices which have been deleted from the modeled system, the deletion being reflected in the *dts* of the vertices in question.

This wealth in insight comes with a price 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., [85,86], may require that certain data has to be discarded after a certain period. On the other hand, remembering each system change causes the RTM^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 elements have to be considered.

For these cases, the remedy lies in knowing when and what to forget. In this section we present the functionality of INTEMPO which allows for the optional constraining of the history representation. This constrained representation contains all elements that are relevant to query executions and, compared to the unconstrained representation, may afford increased memory efficiency.

6.1 RTM^H maintenance based on time window

In case the *dts* of an element is equal to ∞ , the element is still present in the modeled system and thus, due to causal connection, may not be removed from the RTM^H . This is not true for elements 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, elements might be relevant to query executions for a certain period even after they have been deleted from the modeled system. When this period elapses, deleted elements may be considered for removal from the RTM^H .

For example, let $\zeta_2 := (n_{1.1}, \psi_2)$ be a query with $\psi_2 := \Diamond_{[2,5]} \exists n_{1.2}$ —as for ζ_1 it is assumed that the system tracks time in seconds hence time units represent seconds. Assume that ζ_2 is executed over $H_{[7]}$ in Fig. 5. The vertex d_1 has been deleted yet it is relevant to the execution of the query. Given the timing constraint of the temporal operator in ζ_2 , the same would stand for an $H_{[10]}$: although deleted, d_1 might still be involved in the query execution. The involvement of d_1

is *final* at a time point τ , only for RTM^H instances whose associated timestamp exceeds the sum of the right end-point of the timing constraint of $\Diamond_{[2,5]}$ and τ .

Given a (finite) set of MTGCs Ψ , we determine the *time window* $\mathcal{W}$, i.e., the period for which the involvement of *deleted* elements in query executions is not final as follows. First, we compute w for an MTGC $\psi \in \Psi$:

$$w(\psi) = \begin{cases} r(I) + \max(w(\chi), w(\omega)) & \text{if } \psi = \chi \cup_I \omega \\ r(I) + \max(w(\chi), w(\omega)) & \text{if } \psi = \chi \cup_I \omega \\ \max(w(\chi), w(\omega)) & \text{if } \psi = \chi \wedge \omega \\ w(\chi) & \text{if } \psi = \neg \chi \\ w(\chi) & \text{if } \psi = \exists(n, \chi) \\ 0 & \text{if } \psi = \text{true} \end{cases} \quad (9)$$

For Ψ , the time window is given by:

$$\mathcal{W}(\Psi) = \max_{\psi \in \Psi} w(\psi) \quad (10)$$

If the constrained representation option is enabled, INTEMPO performs the computation of $\mathcal{W}$ during Operationalization. Maintenance is performed after Execution and resembles *garbage collection*, i.e., it *prunes* all elements ϵ in an RTM^H whose *dts* exceed a certain threshold. Maintenance yields a pruned RTM^H , denoted by P and defined practically as follows.

$$P_{[\tau_i]} = \begin{cases} H_{[\tau_i]} & \text{if } i = 1 \\ \{\epsilon \mid \epsilon \in H_{[\tau_i]} \wedge \epsilon.dts \geq \tau_{i-1} - 2\mathcal{W}\} & \text{otherwise} \end{cases} \quad (11)$$

where τ_i is the time point of the event e with index i in $\bar{h}_\tau$ (see Sect. 2.2) and $H_{[\tau_i]}$ is the RTM^H corresponding to the coalescence of the e at τ_i and $H_{[\tau_{i-1}]}$. The pruning threshold $\tau_{i-1} - 2\mathcal{W}$ spans a period for which deleted vertices are relevant to the query execution at τ_i , i.e., $[\tau_{i-1} - \mathcal{W}, \tau_i]$. However, to ensure correctness of results, the threshold also covers a preceding period, i.e., $\tau_{i-1} - 2\mathcal{W}$, which is motivated in the following section.

6.2 Projected answer set and intended setting

An $H_{[\tau]}$ constitutes a complete representation of a history of events $\bar{h}_\tau$, i.e., it contains all changes from the beginning of the history up to and including the time point τ . The removal of elements from $P_{[\tau]}$ renders the representation of $\bar{h}_\tau$ partial. In the following it is shown that, over the course of a history, i.e., the sequence of events, all changes to the temporal validity $\mathcal{V}$ of a match can be detected in both representations. However, the $\mathcal{V}$ returned by constrained representations is

Fig. 10 Projected temporal validity of a match obtained for two consecutive time points τ_i and τ_{i+1}

restricted to a certain period to correspond to the representation being partial. 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 $\mathcal{V}$ might have changed (captured by $\mathcal{W}$) since the previous event at τ_{i-1} , that is, the interval $\Gamma = [\tau_{i-1} - \mathcal{W}, \tau_i]$.

Definition 4 (*projected answer set* $\mathcal{T}^\pi$) Given an answer set at time point τ_i over an RTM^H $H_{[\tau_i]}$, denoted by $\mathcal{T}_{\tau_i}(H_{[\tau_i]})$, the *projected answer set* $\mathcal{T}_{\tau_i}^\pi(H_{[\tau_i]})$ is defined as follows.

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

$\mathcal{T}^\pi$ is defined identically for a pruned RTM^H P .

For a match $(m, \mathcal{V})$, all modifications corresponding to an event that could have affected its $\mathcal{V}_{\tau_i}$ are within $[\tau_i - \mathcal{W}, \tau_i]$. $\mathcal{T}_{\tau_{i+1}}^\pi$ contains this interval as well as the interval $[\tau_i, \tau_{i+1}]$. Therefore, with respect to τ_{i+1} , all time points in $\mathcal{V}$ before $\tau_i - \mathcal{W}$ are *final*, i.e., no modification after τ_{i+1} could affect them. This fact also motivates the pruning threshold: deleted elements are kept in P as long as their *dts* is larger or equal to the earliest time point for which a modification could affect the earliest time point for which $\mathcal{V}$ is not final.

See Fig. 10 for an illustration. The $\mathcal{T}_{\tau_i}^\pi$ (the dotted grid) returns a match $(m, \mathcal{V}_{\tau_i})$ where $\mathcal{V}_{\tau_i}$ contains all those time points which are not final with respect to τ_i . Analogously, so does $\mathcal{T}_{\tau_{i+1}}^\pi$ (in gray) with $\mathcal{V}_{\tau_{i+1}}$. At τ_{i+1} however, all time points before the pruning threshold $\tau_i - 2\mathcal{W}$ (that were non-final in $\mathcal{T}_{\tau_i}^\pi$) have become final. An aggregation of $\mathcal{T}_{\tau_i}^\pi$ and $\mathcal{T}_{\tau_{i+1}}^\pi$ would still contain final time points in $\mathcal{V}_{\tau_i}$ even if the pruning of an element at τ_{i+1} would cause these time points to be excluded from $\mathcal{V}_{\tau_{i+1}}$.

Theorem 2 Let $\bar{h}_\tau$ be a history, $\mathcal{D}$ be the last index of $\bar{h}_\tau$, and $\zeta := (n, \psi)$ with $\zeta \in \mathcal{L}_T$. Moreover, let H, P be a complete and a pruned RTM^H , respectively. Over the history, $\mathcal{T}^\pi(H)$ is equal to $\mathcal{T}^\pi(P)$, that is:

$$\bigcup_{i=1}^{\mathcal{D}} \mathcal{T}_{\tau_i}^\pi(H_{[\tau_i]}) = \bigcup_{i=1}^{\mathcal{D}} \mathcal{T}_{\tau_i}^\pi(P_{[\tau_i]}) \quad (12)$$

Proof (sketch) By induction over $\mathcal{D}$. See Sect. B.2 for the proof. $\square$

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 an un-pruned RTM^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 execution times over P .

Moreover, the deletion of elements 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.

6.3 Maintenance with dynamic sets of queries

INTEMPO seamlessly supports non-fixed, i.e., dynamic, sets of queries over a complete RTM^H . The following functionality enables their support by a pruned RTM^H .

Upon an invocation of INTEMPO, the Operationalization checks whether the set of queries has been altered and, if yes, re-computes $\mathcal{W}$. If $\mathcal{W}$ hasn't changed or has been decreased, the invocation proceeds as usual by executing the queries. Else, if $\mathcal{W}$ has increased, the derivation of a complete answer set for the *incoming* queries cannot be ensured for a period of time that is equal to the difference of the previous value of $2\mathcal{W}$ to the newly computed one. In this case, the queries for which a complete $\mathcal{T}$ cannot be guaranteed, are not admitted for execution until the length of the history represented by the RTM^H suffices for their $\mathcal{T}$.

For example, assume INTEMPO is first executed for the previously introduced query ζ_2 with $\Psi = \{\psi_2\}$ where $\mathcal{W} = 5$. Later on, the query ζ_1 is added to the input queries which leads to $\Psi = \{\psi_1, \psi_2\}$. The time point of addition is marked by τ of the RTM^H $H_{[\tau]}$. The alteration induces a recomputation of $\mathcal{W}$ to $\mathcal{W}' = 60$. The value has been increased therefore, based on τ as well as the difference $2\mathcal{W}' - 2\mathcal{W}$, ζ_1 is admitted for execution only when an RTM^H $H_{[\tau']}$ is induced with $\tau' - \tau \geq 2\mathcal{W}' - 2\mathcal{W}$, i.e., in this case, at least 110 time units after its addition.

6.4 Considerations

In most cases, a pruned RTM^H would be more memory-efficient than a complete one. In fact, against a constant rate of incoming events, pruning would yield an RTM^H whose memory consumption would be bounded. However, if the event rate does not have a fixed upper bound, the memory consump-

tion, although mitigated by pruning, would still increase over time.

In the case of MTGCs with an unbounded past operator, i.e., with $r(I) = \infty$, an RTM^H cannot be pruned as it is obvious that the temporal requirement refers to the entire history. In the case of an unbounded future operator, the MTGC may be *non-monitorable* [see 87], i.e., satisfaction of an MTGC may depend on the entire infinite future of the execution thus it can never be provided. Monitoring such formulas is achieved via a practical solution, e.g., the replacement at runtime of the right end-point of the operator with the largest time point of the sequence, which affords a verdict on the satisfaction at any point of monitoring.

Pruning may reduce the size of the matching search space and thus may improve the pattern-matching time of the Execution operation. On the other hand, Maintenance performs two more tasks that should be taken into consideration with respect to the overall performance. First, it uses a priority queue of deleted elements in the RTM^H and iteratively polls the queue to detect and prune elements whose *dfs* exceeded the threshold. Second, every time an element is pruned, Maintenance re-computes the matches maintained by the temporal GDN to detect whether pruned elements have affected any matches and, if so, ensure that affected matches are re-computed based on the latest RTM^H .

7 Runtime monitoring with temporal graph queries

INTEMPO processes a sequence of events which represents an ongoing system behavior and, during query execution, checks whether the observed sequence (captured in the RTM^H) satisfies a temporal logic formula (captured in the MTGC of the query). This functionality resembles the monitoring approach known as Runtime Verification (RV) [see 7]. In this section, we discuss the application of INTEMPO for RV.

7.1 Temporal graph queries for temporal properties

RV represents the system behavior by sequences of states or events at some level of abstraction. An online algorithm is then used to check whether (some prefix of) the sequence satisfies a given property [9], i.e., a formal statement. RV focuses on the verification of *temporal safety properties* [1], i.e., statements of the form “something bad should never happen” which in temporal logic is expressed by prefixing each formula with the *always* operator [62], i.e., the abbreviation of $\neg\Diamond_{[0,\infty)}\neg\psi$ in MTGL with ψ an MTGC. RV searches for violations of such properties which, on a finite sequence, can always be detected as soon as they occur [72].

In the context of graphs, a temporal safety property which contains no temporal operators corresponds to a graph query in $\mathcal{L}$ —see Sect. 2.4. The verification of such properties corresponds to updating a host graph based on a sequence of events and, upon each update, searching the graph for matches of the query pattern [24]. If a match is found, then the property is violated.

Properties with temporal operators require the tracing of a pattern in the host graph over time, i.e., over multiple updates. By the incorporation of a temporal logic such as MTGL into $\mathcal{L}_T$ and the consolidation of updates into an RTM^H , temporal graph queries in $\mathcal{L}_T$ are capable of specifying temporal properties where any matches in an RTM^H returned by INTEMPO constitute violations.

As an example, recall the query $\zeta_1 := (n_1, \psi_1)$ with $\psi_1 := \neg \exists n_{1.1} U_{[0,60]} \exists n_{1.2}$. In order for matches of ζ_1 to return the periods for which there is a violation, that is, 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, i.e., $\zeta'_1 := (n_1, \neg\psi_1)$. Executed over $H_{[7]}$ (Fig. 5), ζ'_1 returns one match: the one involving pm_2 is associated with $[7, \infty)$ —the duration of *until* is empty, its negation is $\mathbb{R}$, and hence the temporal validity $\mathcal{V}$ is computed by $[7, \infty) \cap \mathbb{R}$. It can be inferred that the procedure for patient $\text{pID}=2$, i.e., pm_2 does not conform to the guideline—since its $\mathcal{V}$ is non-empty. Moreover, the procedure is violated from time point 7 onward.

The example above demonstrates an advantage of INTEMPO over traditional RV solutions, as RV solutions typically return a value from the Boolean domain, e.g., *true* or *false*, or, in cases with enhanced expressiveness, some value related to the Boolean domain, e.g., *true*, *probably true*, *false* [12]. Conversely, owing to the computations in Definition 1, INTEMPO returns the duration for which the match is valid, i.e., the violation occurs, which is arguably a more intuitive monitoring outcome and can be further utilized by the system.

A temporal graph query can be obtained for all MTGCs. For an MTGC ψ where the topmost condition is not an existential quantification, a query can be obtained by wrapping ψ in a query looking for an empty pattern, e.g., $(\emptyset, \neg\psi)$. An answer set for $(\emptyset, \neg\psi)$ would consist of an empty match and a temporal validity which would mark the duration for which ψ is violated.

7.2 Postponing a decision

In the previous example, where ζ'_1 is executed over $H_{[7]}$, INTEMPO returns a match for pm_2 , i.e., a violation, although, based on the interval of $\Diamond_{[0,60]}$, 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.

For $H_{[7]}$, the scheme makes a decision based on what has been already observed. Typically, in RV, a decision is not made unless conclusive, i.e., there is no possible future that could satisfy the property. A given RTM^H is sufficient for making a conclusive decision for a temporal property which concerns the past. This is not the case for temporal operators which concern the future where, in order to be conclusive, the decision would have to be postponed. This practice may be particularly useful in systems where humans interoperate with a software system (as in the SHS case-study) and the system acts as a safety net, i.e., only if other planned actions have not been taken within a certain period.

In this section, we extend Execution to return only those matches whose temporal validity has become final since the last time the operation was performed. This functionality is enabled by equipping Execution with a predicate defined as follows.

Definition 5 ($\mathcal{P}$) Let $(n, \neg\psi)$ be a query with n a pattern and ψ an MTGC. Moreover, let $\mathcal{W}$ be the time window of ψ and $\mathcal{V}(m, \neg\psi)$ be a *non-empty* temporal validity of a match m for n in $H_{[\tau]}$ with $\tau \in \mathbb{T}$. Then, the predicate $\mathcal{P}_{m,\tau}$ is defined as:

$$\mathcal{P}_{\tau_i}^m = \begin{cases} \text{true} & \text{if } \mathcal{V}(m, \neg\psi) \cap [\tau_{i-1} - \mathcal{W}, \tau_i - \mathcal{W}] \neq \emptyset \\ \text{false} & \text{otherwise} \end{cases} \quad (13)$$

Intuitively, a true $\mathcal{P}_{\tau_i}^m$ means that there is a time point or more for which the temporal validity has become final with respect to τ_i . The Execution operation checks all matches with a non-empty $\mathcal{V}$ and returns only those for which $\mathcal{P}_{\tau_i}^m$ is true. A decision on time points for which $\mathcal{V}$ is not final is postponed for the next event.

Equipped with $\mathcal{P}_{\tau_i}^m$, an invocation of INTEMPO for $H_{[7]}$ finds the match involving pm_2 and the associated interval $[7, \infty)$. However, the predicate for the match is computed to be false, i.e., intuitively, ψ_1 could be satisfied in the future of $H_{[7]}$, and thus INTEMPO does not return the match. Had there been no drug service added for that patient in future instances of the RTM^H , the match would have started being returned in all RTM^H instances $H_{[\tau']}$ with $\tau' \geq 67$.

The predicate may impose a delay on the detection of some violations which, however, is inherent in reactive monitoring and, in practice, it is often handled by an appropriately timed periodic event generated by the monitor.

In the context of the SHS and similar systems, this functionality enables waiting for a planned action to be taken by a clinician and only detecting a violation after said period has elapsed. The actual period can be set such that the detection serves as a warning, i.e., it anticipates an omission and allows

for some time where the action can still be taken, either by clinicians or smart devices of the SHS.

8 Application scenario: runtime adaptation

This section applies INTEMPO in a self-adaptation scenario which utilizes temporal graph queries and the history-awareness of an RTM^H .

8.1 Self-adaptation based on RTMs

Self-adaptive systems are able to modify their *own behavior* or *structure* in response to their perception of their context, the system itself, and their requirements [31]. Self-adaptation can be generally achieved by adding, removing, and re-configuring components as well as connectors among components in the system architecture [77], therefore, the architecture view is typically considered an appropriate abstraction level, e.g., [46,47]. Causal connection renders RTMs a natural choice for representing the system architecture, as adaptations can be realized as changes on the RTM which are subsequently mirrored in the system [see 107].

An established method of instrumenting self-adaptation is to equip the system with an external feedback control loop, i.e., an *adaptation engine*. An established reference model for the design of an adaptation engine is the *MAPE-K* feedback loop [66]. The MAPE-K loop monitors and analyzes the system and, if needed, plans and executes 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 knowledge. 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.

A self-adaptive system can be adapted via *adaptation rules*. Adaptation rules represent fine-grained units of change that can be performed on the underlying system. The execution 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) [73].

The graph-based encoding of RTMs allows for a realization of adaptation rules in form of graph transformation rules where adaptation issues are expressed via graph patterns which in turn characterize graph queries, i.e., the *LHS* of the rule. Graph queries are executed during analysis and the RTM is adapted via in-place graph transformations based on the *RHS* of the rule [47].

8.2 History-aware self-adaptation via INTEMPO

An RTM^H captures the current state of an RTM as well as temporal information on when changes to said RTM occurred. Furthermore, temporal graph queries are capable of formulating adaptation rules whose conditions include temporal requirements on the history of the system structure. By replacing the RTM in a MAPE-K loop with an RTM^H , INTEMPO can be used to execute temporal graph queries which, in this context, correspond to checking adaptation conditions, over a sequence of events which represent changes on the architecture. The query answers can be used to plan and execute adaptations. Therefore, INTEMPO may serve as the basis for an adaptation engine that enables *history-aware self-adaptation*—see Fig. 11.

Figure 12 depicts a detailed view on an adaptation 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 Fig. 12) is performed during setup. Execution (“E”) is performed in the Analyze activity. The engine features a *Maintain* activity—an extension compared to a canonical MAPE loop—during which Maintenance of INTEMPO (“M”) is performed.

Fig. 11 Overview of INTEMPO and system interaction for adaptation

Fig. 12 History-aware adaptation engine

8.3 Self-adaptation scenario for SHS

In the following, we build on the SHS introduced in Sect. 2.1 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 [78,89], 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*” [78]. Note that, from the point-of-view 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., [110, p. 11]. Therefore, a system adaptation could occur after the deadline is missed and still remedy the situation.

The event log in [78] contains real medical records from a hospital where patients diagnosed with sepsis were treated according to the guideline. The log contains a multitude of events. We 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 (Fig. 2) 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 Fig. 13. To make sure these two actions are referring to the same patient, $g_{1.1}$ contains an attribute constraint (in braces) that checks whether the *pID* of *d* and *pm* are equal.

Based on g_1 and $g_{1.1}$ in Fig. 13, the instruction is formulated in $\mathcal{L}_T$ by the query $MG1 := (g_1, \neg\psi_{MG1})$ with ψ_{MG1} the MTGC $\Diamond_{[0,3600]}\exists g_{1.1}$. That is, the query searches for matches of g_1 , which identifies a (previously untreated) patient with sepsis, for which, in the next hour, there is no match for pattern $g_{1.1}$, which identifies the administration of antibiotics to the *same* patient. The binding of elements in $g_{1.1}$ from g_1 is illustrated in Fig. 13 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 scheme with a more complicated scenario, we also search for violations for a variation of $MG1$. Namely, that no patient with sepsis should be released prior to being treated, a requirement that resembles once more

Fig. 13 Patterns for self-adaptation in SHS—the usage of the same label for vertices denotes that vertices refer to the same element in the RTM^H

conformance checks and refers to *Release* events in the original log. The requirement is captured by the query $MG2 := (g_1, \neg\psi_{MG2})$ with $\psi_{MG2} := \neg\exists g_{1.2} \cup_{[0,3600]}\exists g_{1.1}$. The pattern $g_{1.2}$ is similar to g_1 and depicted in Fig. 13. 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 enriched with the relevant temporal data.

Analyze The activity detects adaptation issues. In this context, these are captured by violations of ψ_{MG1} , i.e., the existence in the RTM^H of structural patterns that reflect sepsis cases (g_1) without associated antibiotics ($g_{1.1}$) within one hour or, similarly, for ψ_{MG2} . Hence we execute (separately) the queries $MG1$ and $MG2$. Note the similarity with the monitoring of temporal properties in Sect. 7.1. Technically, the detection is based on the execution of the temporal GDN which has been 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. Thus, INTEMPO is equipped with the $\mathcal{P}$ predicate in Definition 5 to return only matches whose temporal validity is final.

The matches returned by INTEMPO in this activity constitute adaptation issues, and similar to [47], adaptation-related types (Annotation in Fig. 2) 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 violations are matched, g_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 Fig. 2, are created by transformation rules which respect the RTM^H and are, therefore, capable of setting their *cts* and *dts* appropriately.

Plan and Execute In planning, the engine searches for *sepsis* Probes 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., administer 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, *Maintain* uses the time window obtained by *Operationalization* during setup and prunes the RTM^H , i.e., it removes all elements that have been deleted and their involvement in query executions is final. Following the removal of elements, the GDN is re-executed to update matches.

9 Evaluation

This section presents an implementation of INTEMPO which is evaluated based on a simulation of the SHS case-study and a benchmark for graph-based technologies which simulates the operation of social network. Implementation details are presented in Sect. 9.1. In Sect. 9.2, the implementation is integrated in an adaptation engine, as shown in Fig. 12, and evaluated on the SHS adaptation scenario presented in Sect. 8.2, where adaptation issues are detected by the execution of temporal graph queries over real and synthetic data. Query execution times are compared to an RV tool and a time-aware model indexer in Sect. 9.3. In Sect. 9.4 we evaluate the querying performance of the implementation with data of different sizes, generated by the LDBC Social Network Benchmark. We discuss the results in Sect. 9.5 and present threats to their validity in Sect. 9.6.

9.1 Implementation

Our implementation of INTEMPO is based on the Eclipse Modeling Framework (EMF) [36,101], which is a widespread MDE technology for creating software systems. For pattern matching, we use the *Story Pattern Matcher* [48] using the search plan generation strategy presented in [3]. The Matcher uses *local search* to start the search from a specific element of the graph and thus reduces the pattern matching effort [64]. It uses an OCL checker for checking attribute constraints. For computations on intervals we use an open-source library [52]. 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 poten-

tially expensive shifting of cells in the underlying array list and renders the removal more efficient. The implementation is available as an EMF plugin in [81]. The plugin relies on two domain-specific languages for the generation of an event mapping (see design-time artifacts in Sect. 3) and the specification of temporal graph queries in $\mathcal{L}_T$. For the evaluation, we developed two variants based on the plugin: **IT**, with pruning disabled, and **IT_{+p}**, with pruning enabled.

9.2 Runtime adaptation of smart hospital system

We developed a simulator of the adaptable SHS presented in Sect. 8. The simulation replays events on an RTM^H based on the real and synthesized event logs described in Sect. 9.2.1. After each event, the temporal graph queries MG1 and MG2, described in Sect. 8.2, are executed. Matches constitute adaptation issues which are resolved by appropriate modifications to the RTM^H .

9.2.1 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 [78]. Each trajectory comprises a sequence of events. The events that are relevant to the case-study 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 [33] to generate two synthetic logs, *x10* 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.) 10 and 100 times, respectively, allowing us to test the scalability of INTEMPO without altering the statistical characteristics of the *real* log. The logs and a detailed description of the statistical methods employed are available in [93].

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 10 seconds and one hour, respectively. The logs that contain the deletions are available in [96]. An overview of the logs is shown in Table 1. As shown

Table 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

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

by the *Deleted* column, all created elements are eventually deleted.

9.2.2 Experiment design

We integrated each variant in an adaptation engine. We denote this integration by an arrow circle: $IT^{\odot}$, includes the Monitor, Analyze, Plan, and Execute activity—in terms of INTEMPO, the operations Operationalization and Execution—and $IT_{+p}^{\odot}$, includes all activities above plus Maintenance, i.e., Operationalization, Execution, and Maintenance of INTEMPO. See Fig. 12 for an overview. The experiments³ simulate the events in the *real*, *x10*, and *x100*. Each experiment entails the execution of one variant for one query, either MG1 or MG2.

We measure $IT^{\odot}$ and $IT_{+p}^{\odot}$ 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 $IT_{+p}^{\odot}$, Maintain. The time required for Analyze is effectively the query execution time. Time measurements are used to assess the requirement for fast query executions (R3). 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 requirement for memory-efficiency (R4). The time spent in Monitor, i.e., processing an event, is negligible and thus not reported.

A loop is invoked periodically based on a predefined but modifiable frequency. In our experiments, based on the IAT₇ 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 violation detection, i.e., in the worst case, a violation 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 temporal GDN as well as the time window utilized by Maintenance of $IT_{+p}^{\odot}$, is executed 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 RTM^H: an *ER Sepsis Triage* event corresponds to the addition of a *PMonitoringService* and a *Probe* instance with status set to *sepsis* to the model; an *IV Antibiotics* event 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*. 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 $IT_{+p}^{\odot}$, Maintain is performed and matches are recomputed.

9.2.3 Results

Figures 14 and 15 depict the cumulative time (in logarithmic scale) for each of the measured loop activities and the reaction, 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 executed. The number of processed events in the experiments with the log files *real*, *x10*, and *x100* grows steadily—see Sect. 9.2.1. For $IT^{\odot}$, this increase is mirrored in the size of the RTM^H. The Analyze time of $IT^{\odot}$ increases with respect to these two parameters. The growth of the RTM^H can also be seen in Table 3, where the maximum memory measurement is reported for both variants. Contrary to $IT^{\odot}$, the pruning in $IT_{+p}^{\odot}$ minimizes the size and therefore 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 $IT_{+p}^{\odot}$ increases at a considerably smaller pace compared to $IT^{\odot}$. Note that pruning forces a re-computation of the results. Therefore, as shown in Figs. 14 and 15, the time it requires is non-negligible. Figure 16 shows the time for Analyze for each loop of the two variants for the *x100* log (in logarithmic scale). The pruning of RTM^H enables the analysis time of $IT_{+p}^{\odot}$ to remain constant.

³ All experiments have been conducted on an Intel E5-2643 with 256 GBs of DDR4 RAM and an OpenJDK8 JVM.

Fig. 14 Cumulative time of loop activities for MG1

9.3 Comparison to state-of-the-art

During the Analyze activity, Execution, i.e., the operation of INTEMPO, finds matches by checking whether a temporal property, i.e., an MTGC, is violated by the RTM^H. This functionality resembles the objective of RV, where an online algorithm verifies whether a sequence of events representing a system execution violates 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^U and IT^U_P in detecting issues during the activity.

RV tools are typically not intended for usage with structural models. In MDE, storing a structural model, i.e., a graph, may be achieved by a graph database. However, only few graph databases support a notion of time in the representation, the query specification, and query execution, e.g., the database presented in [59]. None of the time-aware databases supports incremental execution of queries with complex patterns, which renders them sub-optimal for an online setting where the result set is updated after each change to the model.

HAWK is 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 [4]. HAWK has been recently extended to integrate the time-aware database from [59] and to support temporal primitives which can be used to query the history of a model directly. Using these primitives, we formulate the queries MG1 and MG2 and compare the performance of HAWK to that of INTEMPO.

9.3.1 Runtime verification with MONPOLY

MONPOLY [9,10] 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 [34,60] and among

Fig. 15 Cumulative time of loop activities for MG2

Fig. 16 Time for Analyze activity per variant (MG1 - $x100$)

top-performers in an RV competition [8]. Its specification language is based on the Metric First-Order Temporal Logic [9] (MFOTL) 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 violates 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 types of 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 11]. Although the difference in interpretations may affect more extensive evaluations, it does not affect the conditions of the queries discussed in Sect. 8.3.

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 pattern matches only entities with overlapping lifespans, we use a construction suggested by the MONPOLY authors [9]: for a creation event $c(a)$ and a deletion event $d(a)$, we encode the lifespan of the entity a by $\neg \exists d(a) S_{[0,\infty)} \exists c(a)$.

Since the output of MONPOLY focuses on 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.

We map MG1, i.e., $(g_1, \neg \psi_{MG1})$ with ψ_{MG1} the MTGC $\Diamond_{[0,3600]} \exists g_{1.1}$ (see Sect. 8.3), in a straightforward manner to its MFOTL equivalent, i.e., g_1 is enclosed by an existential quantifier, other operators remain intact, and relations are used instead of patterns. This is not possible for $(g_1, \neg \psi_{MG2})$ with $\psi_{MG2} := \neg \exists g_{1.2} U_{[0,3600]} \exists g_{1.1}$ 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 $g_{1.2}$ 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 from above. In the following, we compare to MONPOLY only for MG1. The translations in MFOTL are shown in Appendix C—see Listing 1 and Listing 2.

9.3.2 Indexing and querying an RTM^H with HAWK

HAWK [35,44] integrates a time-aware graph database [59] 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 draws from the well-known *Object Constraint Language* (OCL) [90]. HAWK extends EOL with supports 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 ver-

sions. EOL supports methods native to EMF which obtain the container of an instance or the contents of one of its features.

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. 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 queries are available in Listing 3 and Listing 4 in Appendix C.

9.3.3 Input logs

Regarding MONPOLY, we encoded the SHS metamodel by relations, following generally standard practices [see 88]. 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 1—translations are prefixed with an “M-.” The logs are available in [96].

HAWK supports models created in EMF which allows us to re-use the SHS metamodel in Fig. 2 as well as a part of our implementation to map events in the log files *real*, *x10*, and *x100* to model modifications. The modifications are identical to those created for INTEMPO.

9.3.4 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 Sect. 9.2.2. Each experiment entails checking 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,

Table 2 Issue detection time (cumulative) for MG1 and MG2 (secs - rounded)

	MG1			MG2		
	<i>real</i>	<i>x10</i>	<i>x100</i>	<i>real</i>	<i>x10</i>	<i>x100</i>
IT [○]	7	596	68952	8	585	68744
IT [○] _{+P}	2	10	113	2	14	147
HAWK	730	57685	> 250000	726	57534	> 250000
MONPOLY	2	283	189638	–	–	–

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 execute 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*($\tau - 2\mathcal{W}$), which retrieves all instances from τ up to $2\mathcal{W}$, with τ the timestamp of the latest change and $\mathcal{W}$ the time window—see Sect. 6.1. The latest HAWK version at the time of writing is used (2.2.0) and run on the same machine as INTEMPO. Query execution time and memory consumption are measured similarly to INTEMPO. The time required for saving and committing the XMI file is omitted.

9.3.5 Results

The results for the *issue detection time* (in seconds) for the three logs are shown in Table 2. The issue detection time refers to the amount of time each tool or variant requires to produce the correct result: for IT[○], this is only the sum of the times for *Analyze*, i.e., the query execution time, for every invocation; for IT[○]_{+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 for *x100* was stopped after almost three days, hence no results are reported.

Issue detection with MONPOLY is faster than IT[○] for *real* and *x10*. However, MONPOLY is slower than IT[○] for *x100*. IT[○]_{+P} outperforms MONPOLY for all logs. The reason is that, as shown in Table 3, pruning significantly reduces the size of the model and, therefore, its memory consumption. As a result the time spent for pattern matching is decreased. HAWK is the slowest in detecting issues as well as the most costly in terms of memory. The size of HAWK's database on disk was deemed irrelevant and is not reported.

Figure 17 (in logarithmic scale) shows the *speedup* of IT[○] over HAWK, i.e., (hk/it), with hk the issue detection time of HAWK for an invocation and it the issue detection time of IT[○]

Table 3 Memory consumption (max) for MG1 and MG2 (MBs)

	MG1			MG2		
	<i>real</i>	<i>x10</i>	<i>x100</i>	<i>real</i>	<i>x10</i>	<i>x100</i>
IT [○]	48	268	2479	58	365	3442
IT [○] _{+P}	28	28	28	30	30	30
HAWK	194	1201	–	194	1204	–
MONPOLY	17	39	269	–	–	–

for the same invocation. The speedup value of 1 is marked by a dashed line. IT[○] 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 18 shows the speedup of IT[○]_{+P} over HAWK. HAWK was always slower than IT[○]_{+P}. In fact, their difference increases as the simulation proceeds, as pruning in IT[○]_{+P} decreases the size of the RTM^H. Plot points where the speedup of IT[○] is larger than the speedup of IT[○]_{+P} are invocations in which query execution of IT[○] took a very small amount of time, i.e., less than a millisecond; although the query execution time in these invocations was similar for IT[○]_{+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 speedup comparison was not possible.

9.4 LDBC social network benchmark

The *Social Network Benchmark* (SNB) from the Linked Data Benchmark Council [75] 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. The latest version of the benchmark at the time of writing (0.4.0) generates data which contain both *insert* and *delete* operations [108] and can be conveniently transformed into a stream of timestamped creation and deletion events.

9.4.1 Metamodel and queries

The SNB metamodel consists of a *static* part, i.e., the entities *City*, *Country*, *Tag*, *TagClass*, *University*, and *Company* whose instance creations precede 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 Fig. 19—where entities in gray are explained later in this section. In the generated data, forum memberships and friendships are represented by links between entities.

Fig. 17 Speedup of issue detection time of IT° over HAWK (MG1 - $x10$)—values below the dashed line indicate invocations for which IT° was slower than HAWK

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** [39]. The (slightly adjusted) query IC4 reads: “Given a start Person, find Tags that are attached to Posts that were created by that Person’s friends. Only include Tags that were attached to friends’ Posts created within a year after they became friends with the start Person, and that were never attached to friends’ Posts created before that.” Similarly to the SHS case-study, statements in the query are captured as patterns—shown in Fig. 20.

The query refers to the point in time a friendship was created. Executing 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., [70], links of interest can be encoded as vertices. The links that represent a friendship and a forum membership are relevant to IC4 and IC5 and have been modeled as vertices with a creation and a deletion timestamp in Fig. 19—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 that time is assumed to be tracked in seconds. We search for Tags and Persons ($q_{1.1}$) 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 $\exists(q_{1.1.1}, \neg\Diamond_{(0,\infty)}\exists q_{1.1.1.1})$, where we make use of the knowledge that the `HasMemberLink` will be the last vertex created in $q_{1.1.1}$, i.e., after the

Fig. 18 Speedup of issue detection time of IT°_{+P} over HAWK (MG1 - $x10$)

two Persons. The second condition requires friends to have posted Posts with Tags where the Posts’ first time point (moment of creation) was within the last year: $\Diamond_{[0,1y]}\exists(q_{1.1.2}, \neg\Diamond_{(0,\infty)}\exists q_{1.1.2.1})$ —note that we abbreviate the number of seconds in a year by “1y.” Finally, these Tags have never been attached to Posts of the start Person by any other friend: $\neg\Diamond_{[0,\infty)}\exists(q_{1.1.2.2}, \exists q_{1.1.2.2.1})$.

The construction locating the first time point a pattern occurs, that is, $\exists(q_{1.1.1}, \neg\Diamond_{(0,\infty)}\exists q_{1.1.1.1})$, 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., $\Diamond_{(0,1)}$, 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 $\mathcal{L}_T$, IC4 is captured by the query IC4 := (q_1, ψ_{IC4}) with ψ_{IC4} :

$$\begin{aligned} & \exists(q_{1.1}, \exists(q_{1.1.1}, \exists^f q_{1.1.1.1})) \\ & \wedge \Diamond_{[0,1y]}\exists(q_{1.1.2}, \exists^f q_{1.1.2.1}) \\ & \wedge \neg\Diamond_{[0,\infty)}\exists(q_{1.1.2.2}, \exists q_{1.1.2.2.1})) \end{aligned} \quad (14)$$

The patterns for IC5 are shown in Fig. 21. 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 $\mathcal{L}_T$ it is captured

Fig. 19 Relevant excerpt of the metamodel of the LDBC Social Network Benchmark

by the query $IC5 := (p_1, \psi_{IC5})$ with ψ_{IC5} :

$$\begin{aligned} & (\exists p_{1.1} \vee \exists(p_{1.2}, \exists p_{1.2.1})) \\ & \wedge (\Diamond_{[0,2mo]} \exists(p_{1.3}, \exists^f p_{1.3.1})) \end{aligned} \quad (15)$$

Note that the number of seconds in two months has been abbreviated by “2mo.”

9.4.2 Input logs

We have used the SNB’s data generator 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 nodes and 1.5M edges are created in *sf-0.1*, and 3.2M nodes and 17.3M edges in *sf-1*. The generated data span a period of 10 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 nodes 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* instances, respectively. This brings the total of nodes and edges created by the log to 900K and 3.4M, respectively, for *sf-0.1*. The log for *sf-1* creates 10.2M nodes and 38.4M edges—see the overview in Table 1. The logs are available in [96].

The generated data contains two stages: the *operational* 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).

Fig. 20 Graph patterns used for IC4, where *\$input* denotes an input parameter provided to INTEMPO

9.4.3 Experiment design

We envision a scenario where the results from queries IC4 and IC5 are utilized 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 execute 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 [75, p. 25], we first process a large number of operations (the first 35 months) of the operational stage such that a large starting RTM^H has formed before the queries are executed. We call this the *initial* phase of the experiment. The initial phase corresponds to roughly 800K events in *sf-0.1* and 8.8M events in *sf-1*. The queries are executed 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 1.

Fig. 21 Graph patterns used for IC5

We evaluate the performance of IT (no pruning) and IT_{+P} (with pruning). IT_{+P} is only executed for IC5, as IC4 refers to the entire history and, therefore, contains an unbounded operator. Depending on the log, each query is executed for a different start person. This was done to ensure that the start person would be actively involved in query executions. To choose the start person, we created a list that sorted persons in the network 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-0.1* for the person with *id*=361.

In the initial phase, before the periodic execution begins, the temporal GDN 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 execution 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.

Our attempt to evaluate MONPOLY and HAWK with the SNB logs resulted into considerable practical difficulties. In the SHS case-study, we optimized 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 was impossible as there was 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 were not meant for this setting and using them nonetheless would compromise the evaluation conclusions. Therefore, the tools were excluded from the SNB experiments.

9.4.4 Results

Table 4 shows the query execution times. The initial phase for IC4 and IC5 over *sf-0.1* lasted approx. 18 and 39 seconds. 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 small number of deletions, i.e., less than 5%, occur in the log—see Table 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.

Figure 22 shows the query execution times for IT and IT_{+P} in detail, as well as the number of *relevant nodes* (in thousands) added per period in *sf-1*: relevant nodes are those included in the patterns of the temporal GDN nodes for IC5, i.e., instances of HasMemberLink, Person, KnowsLink, and Post. Generally, larger execution times correspond to rounds with a larger number of relevant nodes. On average, the INTEMPO variants handled 15K additions of relevant nodes per period over *sf-1*. The average query execution time was 5.7 secs with IT and 5.8 secs with IT_{+P} for IC5.

Table 5 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

Table 4 Query execution time (cumulative) for IC4 and IC5 (secs - rounded)

	IC4		IC5	
	<i>sf-0.1</i>	<i>sf-1</i>	<i>sf-0.1</i>	<i>sf-1</i>
IT	9	77	11	169
IT _{+P}	–	–	10	174

Table 5 Memory consumption (max) for IC4 and IC5 (MBs)

	IC4		IC5	
	<i>sf-0.1</i>	<i>sf-1</i>	<i>sf-0.1</i>	<i>sf-1</i>
IT	7040	72389	12631	133880
IT _{+P}	–	–	12158	130384

Fig. 22 Query execution time (IC5 - *sf-1*)

for *sf-0.1* and 44.4GBs for *sf-1*. The rest of the memory in Table 5 was consumed by intermediate results maintained by the nodes of the temporal GDN.

9.5 Discussion

We discuss the evaluation results with respect to the requirement for fast query execution times (*R3*) and memory-efficiency (*R4*), from Section 1. In light of the evaluation, we conclude the discussion with remarks on advantages and limitations of INTEMPO.

9.5.1 Fulfillment of requirements

We presented a method to constrain the history represented in the RTM^H (Sect. 6) where we assume that matches are handled as they are returned and can therefore be removed from the RTM^H . The assumption is justified by relevant use-cases in RV and self-adaptation. We assess the requirement *R3* for fast query executions by (i) a comparison between the issue detection times of IT° , that used an un-constrained representation, and IT_{+P}° , a variant that used a constrained one (ii) a comparison to the issue detection times of 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 correct results, i.e., query execution time for IT° , query execution time plus pruning time for IT_{+P}° , indexing time plus query execution time for HAWK, and execution time for MONPOLY.

Both IT° and IT_{+P}° outperform HAWK and IT_{+P}° also outperforms MONPOLY. IT_{+P}° is slower than IT° and MONPOLY, however it operates on an un-constrained representation, 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 IT° . We can therefore conclude that query answers from INTEMPO are provided fast with respect to 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 approaches.

We assess the requirement for memory-efficiency (*R4*) by a similar comparison. IT_{+P}° consumes smaller amounts of memory than IT° while delivering the same results over the entire history. Therefore, IT_{+P}° is more memory-efficient compared to a variant using an un-constrained representation. Moreover, IT_{+P}° is more memory-efficient than HAWK, as well as than MONPOLY for larger logs, i.e., $x10$, $x100$.

Except for serving as a baseline for the variant with pruning, IT° covers other important use-cases where pruning might be infeasible or undesirable. For instance, the queries of interest might change often and thus time 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-explainability in self-adaptive systems [14] where the system is required to explain adaptation decisions in its entire history—for our example, IT° is faster and more memory-efficient than HAWK, which is intended for such use-cases [45]. 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 execution time of INTEMPO scaled adequately with respect to the smaller and structurally simpler datasets in the SHS case-study. 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. Only nodes that are affected by model modifications are executed which, combined with local search, makes for an incremental evaluation framework which is optimized for fast query executions. 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. Pattern matching may skip big parts of the RTM^H that are unaffected and therefore, in relation to the number of events and added nodes (see Table 1), the query execution times were considerably fast.

On the other hand, given the size of the patterns in SNB queries as well as the number of matches, storing all intermediate results has an adverse effect on memory consumption.

The effect was aggravated by temporal GDNs featuring many Σ MR nodes, i.e., negation, conjunction, *since*, and *until*, whose *LHS* currently contains the maximal context required by parent nodes. Effectively, these nodes re-execute queries that have been already executed and re-store their results. For instance, IC5 features eight such nodes, with some of them redundantly storing matches of elements which are in great quantities in the data, e.g., *Posts*.

9.5.2 Advantages and limitations

Regarding the issue detection by an RV tool like MONPOLY, we note that, although similarities in the use-cases of RV can be observed, so can fundamental differences. INTEMPO relies on an RTM, i.e., a causally connected snapshot of the system constituents and their state, which we assume to be the product of a broader MDE context. Queries over the RTM (and their answers) are supposed to be further utilized at once within that context—as the self-adaptation application scenario in Sect. 8 demonstrated. Models are typically not the primary focus of RV tools; contrary to an RTM, representations of the system state are created ad hoc and are typically inaccessible by other tools or end-users, which may render RV tools impractical for use in an MDE context, e.g., it may hinder synergy with other model-based technologies.

Moreover, as mentioned previously, structural RTMs are amenable to a graph-based encoding. Transferring this setting to an adequately expressive RV representation like relations resulted into overly technical and error-prone encodings as, typically, RV tools do not focus on structural representations. Consequently, emulating pattern matching of queries which were rather structurally simple was also cumbersome, even after optimizations. 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 sub-optimal for graph-based representations, queries and, in turn, structural adaptation of systems.

In the presented experimental setting, INTEMPO outperformed HAWK and MONPOLY. Nonetheless, the level of maturity of the two tools surpasses the prototypical implementation we present. HAWK supports various modeling technologies, including EMF. It also seamlessly supports 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, 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 execution and monitoring whether the sum exceeds a limit. We plan to address this limitation by equipping the temporal GDN

with auxiliary nodes used to encode input parameters as constraints that are checked during pattern matching, as shown in our previous work [100].

As demonstrated by the SNB experiments, INTEMPO requires that attributes, types, and links whose evolution is of interest are encoded as vertices. For the SNB, this meant that two links were encoded as vertices, 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 physical time references are not currently supported by $\mathcal{L}_T$ —and are typically not provided by logic-based languages, e.g., MFOTL. HAWK is able to express such constraints referring to physical time. A related limitation is that INTEMPO currently cannot relate the *current* time point, i.e., when the query is executed, with temporal computations by the temporal GDN. Hence, INTEMPO performs an additional check (predicate $\mathcal{P}$) to assess whether query results are conclusive with respect to the current time point. This check is standard in RV and hence seamlessly provided by MONPOLY. We plan to integrate the check in INTEMPO by means of the auxiliary nodes mentioned above.

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 Sect. 9.4. 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 of the SNB queries have been omitted from our formulation of IC4 and IC5 in $\mathcal{L}_T$.

The SNB experiments indicated that larger temporal GDNs may require a significant amount of memory. We plan to investigate whether the patterns used for Σ MR nodes can be optimized so that memory consumption is reduced.

9.6 Threats to validity

We organize this section based on the types of validity in [111], i.e., *conclusion*, *internal*, *construct*, and *external*.

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 [93].

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 execution 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, for the SNB evaluation, we reported on the number of total additions of relevant nodes instead of the number of events per period.

Internal Validity In this context, threats to internal validity are influences which might have affected metrics, i.e., the query execution 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 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.

Construct Validity Threats to construct validity refers to situations where the used metrics do not actually measure the *construct*, i.e., concept. We used the standard metrics of query execution time and memory consumption, measured in conventional ways. We have ensured that these metrics

corresponded to the correct results. First, the detection of violations in the logs by the variants in the SHS evaluation has been manually double-checked by the authors 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 correctness of computation of temporal validity and the detection of all changes to temporal validity despite pruning the history representation.

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 [109], 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 [89].

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 vertices specific to INTEMPO, i.e., the *MonitorableEntity*, or specific to the evaluation, i.e., links of interest were encoded as vertices. We made minor modifications to the data which fixed a small number of consistency issues, i.e., deletions occurring before creations, which probably stem from the fact that deletions in the benchmark's output are a rather new feature [108].

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 interconnections which resembled real social networks. The SHS queries showed the merits of a graph-based temporal logic for graph structures in runtime monitoring scenarios, 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 implementations. 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.

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. 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 in the tool's website [35] 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.

10 Related work

INTEMPO consolidates model instances into a single model, the RTM^H, and answers queries with soft⁴ real-time timing constraints on the history of the RTM^H. The RTM^H is encoded as a graph and hence queries are captured by temporal graph queries which are executed incrementally. We first discuss work that relates to INTEMPO on the technical level of graphs, graph queries, and incremental query executions. Then, we relate INTEMPO to other work from the MDE community that can, either directly or indirectly, query the history of a model. Finally, we discuss relevant work from the

RV community and a related approach to processing streams of events.

Graphs, Queries, and Incremental Execution Formally, INTEMPO is founded on typed attributed graph transformation rules [37]. Therefore, we limit the discussion to graphs although the model and model transformation rules could be encoded in other data models and related technologies, e.g., [65] and [76]. In the RTM^H, time is captured by vertex attributes which are assumed to be present for each vertex in the metamodel, i.e., the type graph. Query executions are performed via graph transformation rules which write or perform computations with these (distinguished) attributes. This rule behavior resembles the foundational approach presented in [54] where graph transformation rules are extended with a notion of time.

The VIATRA framework [104,105] stores a graph in memory and features a query execution engine which executes 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 execution framework. VIATRA uses a decomposition strategy similar to a GDN called RETE [42], whose performance effect on INTEMPO we plan to investigate. A VIATRA extension [102] distributes the pattern matching effort for sub-queries over multiple processing units, which is another interesting future direction for INTEMPO.

The setting assumed by VIATRA is similar to the one of INTEMPO and resembles *streaming* [97] and *active model transformations* [13]: the graph (typically representing an RTM) is constantly being updated by a stream of graph elements or events that are mapped to graph elements; queries are executed after each change and their results are updated. Based on this setting, the VIATRA-based solution Búr et al. by [24] captures safety properties via graph queries and is, therefore, capable of making assertions about the history of the graph. However, the solution does not support the integration of temporal statements into properties; properties only check for the occurrence of prohibited structures. Moreover, the solution does not support incremental query execution.

Implementations of *time-aware graph databases* typically build on a database back-end and introduce extensions which integrate a notion of time, e.g., [32,59,79]. Although 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 [2,98] and, thus, of providing a certain degree of support for reactive settings, none of the implementations supports incremental execution of queries with complex patterns.

The approach presented in [6] analyzes a host graph stored into 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 execution of the query over the sub-graph, which

⁴ We refer to [23] for a discussion on hard real-time constraints in executions of graph queries.

returns a complete answer set. The analysis does not integrate a notion of time—if it were to be extended so as to support temporal primitives and graphs with 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.

Solutions for *graph stream analytics*, e.g., [25,58,67,71,82] may also be seen as related to INTEMPO, since they are also concerned with the efficient storage and querying of graphs that are constantly updated. These solutions model 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 query executions. Moreover, models are not the primary focus of these solutions which may render their utilization in an MDE context cumbersome.

The languages presented by Röttschke and Schürr [91] and in our earlier work [68] build on Story Diagrams [112], a visual language for the specification of pattern-matching tasks, to specify temporal graph queries with timing constraints. The two languages however focus purely on specification and no tool support for either language exists.

Querying History in Model-driven Engineering 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. [56]. 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 could potentially be supported by an RTM^H, it is beyond our current scope. 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 execution of pattern-based queries that refer to a period of time—as Haeusler et al. acknowledge.

Solutions which focus on the history of an RTM typically use an on-disk database to store history and then facilitate the execution of database queries at runtime. HAWK [44], to which INTEMPO was compared, uses the time-aware graph database in [59]. The solutions in [51] and [80] use a map- and time-series-database, respectively. As the evaluation in Sect. 9 indicated, disk 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 temporal GDN in INTEMPO. Compared to an in-memory representation, on-disk databases offer increased scalability with respect to the size of the models that can be stored.

All solutions above query history by means of OCL extensions that support temporal primitives—although HAWK’s extension is more expressive. They also conveniently support attribute-level history (contrary to INTEMPO which requires for attributes to be encoded as nodes in the metamodel, see Sect. 9.5.2) and on-the-fly operations on attribute values, e.g., aggregation. However these features also increase the number of disk accesses during pattern matching.

Runtime Verification and Related Approaches Seen in a broader context, INTEMPO processes a sequence of events and verifies whether this sequence satisfies a temporal logic formula. This approach to verification is known as *Runtime Verification* (RV). In RV, an online algorithm incrementally processes a sequence of timestamped events and checks whether the prefix of the sequence processed so far satisfies a temporal logic property. The algorithm only stores data that are relevant to the property.

Based on this similarity, we compared the performance of INTEMPO to MONPOLY [9,10], a state-of-the-art RV tool. MONPOLY is a command-line tool which notably combines a specification language which is adequately expressive for the use-cases discussed in this article with an efficient incremental monitoring algorithm. Its specification language is based on the Metric First-Order Temporal Logic [9] (MFOTL) which uses first-order *relations* to capture system entities and their relationships.

Besides MONPOLY, solutions stemming from RV typically provide no or only partial support for key features of INTEMPO, e.g., events containing data, native or indirect support for graphs and bindings, and temporal operators with timing constraints: the solution by Dou et al. [34] presents a pattern-based RV technique which concerns propositional events, i.e., containing no data, and is thus unsuitable for use-cases of interest; a monitoring algorithm presented in [11] involves an interval computation similar to ours but concerns a propositional, past-only logic.

An exception is the tool DEJAVU by Havelund and Peled [63] which 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 would be deprived of the one-to-one mapping of temporal operators in $\mathcal{L}_T$ and MFOTL and could be significantly less compact [61,74]. Additionally, Havelund and Peled report that, currently, only timing constraints which span approx. 60 time units or less yield acceptable performance [62]. This restriction renders DEJAVU unsuitable for the application scenarios targeted by INTEMPO.

The approach known as *Complex Event Processing* (CEP) focuses on processing a stream of events and detects temporal patterns based on the content and the ordering of

events. Detected patterns generate *complex events* which form another stream and can be further processed [see 29]. Outwardly, the objective of RV is similar to that of CEP. However, the two 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 clear semantics [53]) which makes a direct comparison difficult; the capability of CEP to evaluate sequential patterns is typically limited, while of central importance to RV [57]—we refer to [57] for a more detailed discussion. Therefore, RV is considerably more relevant to INTEMPO.

Compared to INTEMPO, graph-based solutions that use CEP are limited in their ability to query the history of a model at runtime. Dávid et al. present a solution based on a streaming transformation of an RTM. The solution generates events when patterns are matched and then uses CEP to check whether generated events occur within a given time window, thus capturing, albeit indirectly and to a certain extent, temporal requirements on matched patterns [30]. The solution by Barquero et al. [5] 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.

11 Conclusion and future work

Following the common practice of representing a runtime model as a typed attributed graph, we introduced a language for graph-based model queries which incorporates a temporal logic to enable the formulation of temporal graph queries. Moreover, we introduced a scheme which enables the incremental execution of temporal graph queries over runtime models with history, i.e., runtime models with a creation and deletion timestamp for each entity. The scheme offers the option to prune entities from the model that are not relevant to query executions, therefore reducing the memory footprint. By incorporating a temporal logic, the scheme is capable of monitoring temporal properties at runtime. Building on this capability, we apply the scheme in a runtime adaptation scenario where query matches in the model capture adaptation issues which are handled by in-place model transformations. We present an implementation which we evaluate based on a simulation of the scenario with real and synthetic data. We compare the performance of the implementation in detecting issues to a relevant runtime monitoring tool and a tool from the MDE community which is capable of querying the history of a model. Moreover, we evaluate the implementation with data which simulate the operation of a social network and are generated by an independent benchmark.

In the future, we plan to compare the query decomposition strategy in the framework of INTEMPO with the performance of other strategies. Moreover, we will investigate the number of events per second that INTEMPO can handle in other evaluation settings, which will indicate whether our scheme can be used for different application scenarios with considerably larger event streams. We will also consider technical optimizations such as the usage of indexing structures that can index matches based on their intervals and the optimization of patterns in auxiliary nodes of the evaluation framework. We plan to experiment with more aggressive pruning strategies for application scenarios with limited resources. An interesting research direction is the combination of INTEMPO with an EMF database which stores very old parts of the model and enables on-demand loading for the parts in question. A direction which holds vast potential is the opportunity to observe and learn from the history recorded in the model. Such knowledge could be subsequently used for predictions. Finally, we plan to utilize the ability of the scheme to detect the amount of time remaining for the violation of a property and enable more sophisticated decision-making for runtime adaptations.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

A Intervals

An *interval* is a non-empty, convex set $I \subseteq \mathbb{R}$ and has one of the following forms: (τ, τ') , $[\tau, \tau')$, $(\tau, \tau']$, $[\tau, \tau']$ where (i) τ, τ' is the *left*, respectively *right*, end-point of the interval (ii) $\tau \leq \tau'$ (iii) $\tau, \tau' \in \mathbb{T}$. 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, right end-point of I by $\ell(I)$, respectively, $r(I)$. For example, the interval $I = \{t \mid t \in \mathbb{T} \wedge 0 \leq t < 5\}$, also denoted as $[0, 5)$, 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*, i.e., $\text{adj}_\ell(k, l)$ when $r(k) = \ell(l)$ and k is right-open and l is left-closed; right-adjacent intervals are defined symmetrically and denoted by adj_r . The two intervals are *overlapping*, i.e., $\text{ov}(k, l)$, when $k \cap l \neq \emptyset$. 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 and interval subtraction in a standard way [83]. Let I, k, l be interval 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 1, we rely on the distributivity of an interval addition or subtraction with the regular set intersection, captured by the lemma below.

Lemma 1 *Let $K, L, I \in \mathcal{I}$ and $\circ \in \{\oplus, \ominus\}$. If $K \cap L \neq \emptyset$, $(K \circ I) \cap (L \circ I) = (K \cap L) \circ I$.*

Proof Note that, for the interval I , its negation $-I$ 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 [83], 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 get $\tau \in (K \cap L) \oplus I$.

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. $\square$

B Proofs

B.1 Theorem 1

Additionally to a set of convex sets of time points, i.e., intervals, an interval set can be seen simply as a set of time points. In the following, when we consider an interval set as a set of time points, we underline the interval set, e.g., $\underline{\mathcal{Z}}$. Following is the proof for Theorem 1.

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: By the semantics of MTGL, *true* is always satisfied, hence $\mathcal{Y}(m, \text{true}) = \mathcal{Z}(m, \text{true}) = \mathbb{R}$.

Induction step: $\psi = \exists(\hat{n}, \chi)$. Assume that $\mathcal{Y}(\hat{m}, \chi) = \mathcal{Z}(\hat{m}, \chi)$. We first show that:

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

Let $\tau \in \mathcal{Y}(m, \exists(\hat{n}, \chi))$. By the semantics, a match $\hat{m}$ for $\hat{n}$ exists at τ such that $\hat{m}$ is compatible with the enclosing match m , $\hat{m}$ satisfies χ , and $\max_{e \in E} e.cts \leq \tau < \min_{e \in E} e.dts$, with E the elements of $\hat{m}$. By definition of $\mathcal{Y}$ and λ , it follows that (i) $\tau \in \mathcal{Y}(\hat{m}, \chi)$ and by the premise $\tau \in \mathcal{Z}(\hat{m}, \chi)$ (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 $\tau \in \mathcal{Y}(\hat{m}, \exists(\hat{n}, \chi))$ 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 $\max_{e \in E} e.cts \leq \tau < \min_{e \in E} e.dts$ and $\hat{m}$ satisfies χ . Ergo, $\tau \in \mathcal{Y}(m, \exists(\hat{n}, \chi))$.

By showing inclusion in both directions, we have shown the sets of time points to be equal.

Induction step: $\psi = \chi \cup_I \omega$ 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:

$$\mathcal{Y}(m, \chi \cup_I \omega) \subseteq \bigcup_{i \in \mathcal{Z}(m, \omega), j \in J_i} j \cap ((j^+ \cap i) \ominus I)$$

Let $\tau \in \mathcal{Y}(m, \chi \cup_I \omega)$. 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 premise, $i' \in \mathcal{Z}(m, \omega)$. 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 $[\tau, \tau']$ cannot be empty: From $\tau \in \tau' \ominus I$, it follows that $\tau' \in \tau \oplus I$ and, since $\ell(I) \neq 0$ (and by $I \subseteq \mathbb{T}$, $0 \notin I$), $\tau' > \tau$. Therefore there exists some $j' \in \mathcal{Y}(m, \chi)$ and, by the premise, 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 $\tau \in j'^+ \oplus I$. Finally, from $\tau \in j'$ and $\tau \in j'^+ \oplus I$, we obtain, $\tau \in j' \cap (j'^+ \oplus I)$. Since $\tau \in i' \oplus I$, we obtain $\tau \in j' \cap (j'^+ \oplus I) \cap (i' \oplus I)$ and, by $j'^+ \cap i' \neq \emptyset$ and Lemma 1, $\tau \in j' \cap ((j'^+ \cap i') \oplus I)$. 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 left-adjacent or overlapping intervals for some $i \in \mathcal{Z}(m, \omega)$.

We proceed with showing inclusion $\supseteq$. Let $\tau \in j' \cap ((j'^+ \cap i') \oplus I)$ 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 $\tau \in (j'^+ \cap i') \oplus I$. 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 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, $\chi_{U_I} \omega$ is satisfied at τ and, by definition of the satisfaction span, $\tau \in \mathcal{Y}(m, \chi_{U_I} \omega)$.

By showing inclusion in both directions, we have shown the sets of time points to be equal.

Induction step: $\psi = \chi_{U_I} \omega$ and $\ell(I) = 0$. Assume that the premise from the previous step holds. We first show:

$$\mathcal{Y}(m, \chi_{U_I} \omega) \subseteq \bigcup_{i \in \mathcal{Z}(m, \omega)} i \cup \bigcup_{j \in J_i} j \cap ((j^+ \cap i) \oplus I)$$

Let $\tau \in \mathcal{Y}(m, \chi_{U_I} \omega)$. By definition of the satisfaction span, $\chi_{U_I} \omega$ 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 $\tau \in j' \cap ((j'^+ \cap i') \oplus I)$ 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 $\tau \in i' \cup (j' \cap ((j'^+ \cap i') \oplus I))$. It follows that τ is also a member of $\bigcup_{i \in \mathcal{Z}(m, \omega)} i \cup (\bigcup_{j \in J_i} j' \cap ((j'^+ \cap i') \oplus I))$.

We proceed with inclusion $\supseteq$. Let $\tau \in i' \cup (j' \cap ((j'^+ \cap i') \oplus I))$ for some pairing of $i' \in \mathcal{Z}(m, \omega)$ and $j' \in J_i$ with J_i defined as previously. It follows that $\tau \in i' \vee \tau \in j' \cap ((j'^+ \cap i') \oplus I)$. For the left disjunct, since $0 \in I$, there is 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 $\tau \in \mathcal{Y}(m, \chi_{U_I} \omega)$ 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: $\psi = \chi_{S_I} \omega$ and $\ell(I) \neq 0$. The step proceeds analogously to *until* and relies on the same premise. We first

show:

$$\mathcal{Y}(m, \chi_{S_I} \omega) \subseteq \bigcup_{i \in \mathcal{Z}(m, \omega), j \in J_i} j \cap ((^+ j \cap i) \oplus I)$$

Let $\tau \in \mathcal{Y}(m, \chi_{S_I} \omega)$. 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 premise, 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 $(\tau', \tau]$ cannot be empty: From $\tau \in \tau' \oplus I$, it follows that $\tau' \in \tau \oplus I$ and, since $\ell(I) \neq 0$ (and by $I \subseteq \mathbb{T}$, $0 \notin I$), $\tau > \tau'$. Therefore there exists some $j' \in \mathcal{Y}(m, \chi)$ and, by the premise, $j' \in \mathcal{Z}(m, \chi)$, 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 $\tau' \in \tau \oplus I$, 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, $\chi_{S_I} \omega$ is satisfied at τ and, by definition of the satisfaction span, $\tau \in \mathcal{Y}(m, \chi_{S_I} \omega)$.

By showing inclusion in both directions, we have shown the sets of time points to be equal.

Induction step: $\psi = \chi_{S_I} \omega$ and $\ell(I_U) = 0$. Equality can be shown analogously to the corresponding step for *until*.

From the base case and induction steps, it follows that Theorem 1 holds. $\square$

B.2 Theorem 2

Following is the proof for Theorem 2, i.e.,

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

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 $\subseteq$. The proof proceeds by tracking the match of a tuple and its temporal validity in the projected answer set of a non-pruned RTM^H, via three steps, in (1) the regular answer set over the non-pruned RTM^H, and from there to (2) the regular answer set of a pruned RTM^H, and from there to (3) the projected answer set of the pruned RTM^H.

Let t be a tuple $(m, \mathcal{V}) \in \bigcup_{i=1}^{j+1} \mathcal{T}^\pi(H_{[\tau_{j+1}]})$ for the query (n, ψ) . The tuple t was either in $\bigcup_{i=1}^j \mathcal{T}^\pi(H_{[\tau_j]})$ or not. In the former case, t must have also been in $\bigcup_{i=1}^j \mathcal{T}^\pi(P_{[\tau_j]})$ (from the premise), and therefore, it is also in the union $\bigcup_{i=1}^{j+1} \mathcal{T}^\pi(P_{[\tau_{j+1}]})$.

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 $\mathcal{V}_{t'} \cap [\tau_j - \mathcal{W}, \tau_{j+1}] = \mathcal{V}_t$ with $\mathcal{W}$ the time window of ψ from the query. Step 1 is done.

Henceforth, we denote the interval $[\tau_j - \mathcal{W}, \tau_{j+1}]$ by Γ . By definition of $\mathcal{W}$, an element ϵ may affect a temporal validity at an arbitrary time point x only if $x \in \epsilon.\lambda \ominus \mathcal{W}$ with λ the lifespan of ϵ , i.e., if $\epsilon.dts \geq x - \mathcal{W}$. 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 ϵ with $\epsilon.dts \geq \tau_j - 2\mathcal{W}$ or, since the current time point is τ_{j+1} , all ϵ with $\epsilon.dts \in [\tau_j - 2\mathcal{W}, \infty)$ may affect $\mathcal{V}_{t'}$. By definition of a pruned RTM^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.

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, $t''' = t$; thus, t is also 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 $\supseteq$. Let t be a tuple $(m, \mathcal{V}) \in \bigcup_{i=1}^{j+1} \mathcal{T}^\pi(P_{[\tau_{j+1}]})$ for the query (n, ψ) . If the tuple t was 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. 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 $\mathcal{V}_{t'} = \mathcal{V}_t \cap \Gamma$. By definition of pruning, all elements ϵ which were pruned in $P_{[\tau_{j+1}]}$ yet contained in $H_{[\tau_{j+1}]}$ must have a $dts < \tau_i - 2\mathcal{W}$, i.e., $\epsilon.dts \in [0, \tau_i - 2\mathcal{W})$. 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 $\mathcal{V}_{t''} = \mathcal{V}_{t'}$. 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 $\bigcup_{i=1}^{j+1} \mathcal{T}^\pi(H_{[\tau_{j+1}]})$. We have shown inclusion $\supseteq$. By the base case and the induction step, it follows that the theorem holds. $\square$

C Evaluation supplement

This section presents the specification of queries in MG1 and MG2 in the specification languages of MONPOLY and HAWK, the two tools used for comparisons with INTEMPO in Sect. 9.

C.1 Queries in MFOTL

MONPOLY uses a specification language which is based on the Metric First-Order Temporal Logic (MFOTL). As MTGL, MFOTL is a metric temporal logic, which allows for preserving the main structure of the MTGC MG1. On the other hand, lifespans and patterns require a special handling which we described in Sect. 9. The query MG1 in MFOTL is showed in Listing 1. Note that 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.

```

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 1 The query MG1 in MFOTL

Note that a temporal operator with no metric interval is a shorthand for the same operator with 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 shown in Listing 2. 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 2 The (non-monitorable) query MG2 in MFOTL

C.2 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.

```
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 3 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 4 The query MG2 in EOL

References

- Alpern, B., Schneider, F.B.: Recognizing safety and liveness. *Distrib. Comput.* **2**(3), 117–126 (1987). <https://doi.org/10.1007/BF01782772>
- Apache Foundation (n.d.) EventStrategy - Apache TinkerPop Reference Documentation. https://tinkerpop.apache.org/docs/3.5.1/reference/#_eventstrategy, Accessed on 23 Oct 2021
- Barkowsky, M., Giese, H.: Hybrid search plan generation for generalized graph pattern matching. *J. Log. Algebraic Methods Program.* (2020). <https://doi.org/10.1016/j.jlamp.2020.100563>
- Barmpis, K., Shah, S., Kolovos, D.S.: Towards incremental updates in large-scale model indexes. In: Taentzer, G., Bordeleau, F. (eds.) *Modelling Foundations and Applications*, pp. 137–153. Springer International Publishing, Cham (2015)
- Barquero, G., Burgueño, L., Troya, J., Vallecillo, A.: Extending complex event processing to graph-structured information. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Association for Computing Machinery, New York, NY, USA, MODELS '18*, pp. 166–175 (2018). <https://doi.org/10.1145/3239372.3239402>
- Barquero, G., Troya, J., Vallecillo, A.: Improving query performance on dynamic graphs. *Softw. Syst. Model.* **20**(4), 1011–1041 (2021). <https://doi.org/10.1007/s10270-020-00832-3>
- Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) *Lectures on Runtime Verification: Introductory and Advanced Topics. Lecture Notes in Computer Science*, pp. 1–33. Springer International Publishing, Cham (2018)
- Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *Int. J. Softw. Tools Technol. Transf.* **21**(1), 31–70 (2019). <https://doi.org/10.1007/s10009-017-0454-5>
- Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. *J. ACM* **62**(2), 15:1–15:45 (2015)
- Basin, D., Klaedtke, F., Zalinescu, E.: The Monpoly monitoring tool. In: Reger, G., Havelund, K. (eds.) *RV-CuBES 2017. An*

- International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA, EasyChair, vol. 3, pp. 19–28. Kalpa Publications in Computing (2017)
11. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for monitoring real-time properties. *Acta Inf.* **55**(4), 309–338 (2018). <https://doi.org/10.1007/s00236-017-0295-4>
 12. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşiran, S. (eds.) *Runtime Verification*, pp. 126–138. Springer, Berlin, Heidelberg (2007)
 13. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Active operations on collections. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) *Model Driven Engineering Languages and Systems*, pp. 91–105. Springer, Berlin, Heidelberg (2010)
 14. Bencomo, N., Welsh, K., Sawyer, P., Whittle, J.: Self-explanation in adaptive systems. In: *Proceedings of the 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, USA, ICECCS '12, pp. 157–166 (2012)
 15. Bencomo N, France RB, Cheng BHC, Aßmann U (eds) *Models@run.time: Foundations, Applications, and Roadmaps. Programming and Software Engineering, Lecture Notes in Computer Science, State-of-the-Art Surveys*, Springer International Publishing, (2014)
 16. Bencomo, N., Götz, S., Song, H.: Models@run.time a guided tour of the state of the art and research challenges. *Softw. Syst. Model.* **18**(5), 3049–3082 (2019)
 17. Benelallam, A., Hartmann, T., Mouline, L., Fouquet, F., Bourcier, J., Barais, O., Traon, Y.L.: Raising time awareness in model-driven engineering: Vision Paper. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 181–188 (2017). <https://doi.org/10.1109/MODELS.2017.11>
 18. Beyhl, T., Blouin, D., Giese, H., Lambers, L.: On the operationalization of graph queries with generalized discrimination networks. In: Echahed, R., Minas, M. (eds.) *Graph Transformation*, pp. 170–186. Springer International Publishing, Cham (2016)
 19. Bézivin, J., Paige, R.F., Aßmann, U., Rumpe, B., Schmidt, D.C.: 08331 Manifesto - model engineering for complex systems. In: Aßmann, U., Bézivin, J., Paige, R., Rumpe, B., Schmidt, D.C. (eds.) *Perspectives Workshop: Model Engineering of Complex Systems (MECS)*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, no. 08331 in Dagstuhl Seminar Proceedings (2008)
 20. Biermann, E., Ehrig, H., Ermel, C., Golas, U., Taentzer, G.: Parallel independence of amalgamated graph transformations applied to model transformation. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of His 65th Birthday. Lecture Notes in Computer Science*, pp. 121–140. Springer, Berlin, Heidelberg (2010)
 21. Blair, G., Bencomo, N., France, R.B.: Models@run.time. *Computer* **42**(10), 22–27 (2009). <https://doi.org/10.1109/MC.2009.326>
 22. Brambilla, M., Cabot, J., Wimmer, M.: *Model-driven software engineering in practice*. Second Edition. Synth. Lect. *Softw. Eng.* **3**(1), 1–207 (2017). <https://doi.org/10.2200/S00751ED2V01Y201701SWE004>
 23. Búr, M., Varró, D.: Towards WCET estimation of graph Queries@run.time. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 233–238 (2019). <https://doi.org/10.1109/MODELS.2019.00007>
 24. Búr, M., Szilágyi, G., Vörös, A., Varró, D.: Distributed graph queries over models@run.time for runtime monitoring of cyber-physical systems. *Int. J. Softw. Tools Technol. Transf.* **22**(1), 79–102 (2020). <https://doi.org/10.1007/s10009-019-00531-5>
 25. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: *Proceedings of the 7th ACM European Conference on Computer Systems, Association for Computing Machinery, New York, NY, USA, EuroSys '12*, pp. 85–98 (2012). <https://doi.org/10.1145/2168836.2168846>
 26. Combi, C., Gambini, M., Migliorini, S., Posenato, R.: Modelling temporal. Data-centric medical processes. In: *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium, ACM, New York, NY, USA, IHI '12*, pp. 141–150 (2012). <https://doi.org/10.1145/2110363.2110382>
 27. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: *Handbook of Graph Grammars and Computing by Graph Transformation. Foundations*, pp. 313–400. World Scientific Publishing Co Inc, USA (1997)
 28. Cruz, J.G., Sadovkyh, A., Truscan, D., Bruneliere, H., Pierini, P., Muñoz, L.L.: MegaM@Rt2 EU Project: open source tools for mega-modelling at runtime of CPSS. In: Ivanov, V., Kruglov, A., Masyagin, S., Sillitti, A., Succi, G. (eds.) *Open Source Systems*, pp. 183–189. Springer International Publishing, Cham (2020)
 29. Cugola, G., Margara, A.: Processing flows of information from data stream to complex event processing. *ACM Comput. Surv.* **44**(3), 15:1–15:62 (2012). <https://doi.org/10.1145/2187671.2187677>
 30. Dávid, I., Ráth, I., Varró, D.: Foundations for streaming model transformations by complex event processing. *Softw. Syst. Model.* **17**(1), 135–162 (2018). <https://doi.org/10.1007/s10270-016-0533-1>
 31. de Lemos, R., Giese, H., Müller, H.A., Shaw, M., Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G. (eds.): *Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science*, vol. 7475. Springer, Berlin Heidelberg (2013)
 32. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. *VLDB J.* **30**(5), 825–858 (2021). <https://doi.org/10.1007/s00778-021-00675-4>
 33. DiCiccio, T.J., Efron, B.: Bootstrap confidence intervals. *Stat. Sci.* **11**(3), 189–228 (1996). <https://doi.org/10.1214/ss/1032280214>
 34. Dou, W., Bianculli, D., Briand, L.: 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*, IEEE Press, Austin, Texas, MODELS '17, pp. 323–333 (2017). <https://doi.org/10.1109/MODELS.2017.9>
 35. Eclipse Foundation (n.d.a) Eclipse Hawk. <https://www.eclipse.org/hawk/>, Accessed on 23 Oct 2021
 36. Eclipse Foundation (n.d.b) Eclipse Modeling Framework (EMF). <https://www.eclipse.org/modeling/emf/>, Accessed on 23 Oct 2021
 37. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) *Graph Transformations*, pp. 161–177. Springer, Berlin, Heidelberg (2004)
 38. Elhabbash, A., Bahsoon, R., Tino, P., Lewis, P.R.: Self-adaptive volunteered services composition through stimulus- and time-awareness. In: *2015 IEEE International Conference on Web Services*, pp. 57–64 (2015). <https://doi.org/10.1109/ICWS.2015.18>
 39. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The LDDB social net-

- work benchmark: interactive workload. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA, SIGMOD '15, pp 619–630 (2015). <https://doi.org/10.1145/2723372.2742786>
40. Esfahani, N., Yuan, E., Canavera, K.R., Malek, S.: Inferring software component interaction dependencies for adaptation support. *ACM Trans. Auton. Adapt. Syst.* **10**(4), 26:1–26:32 (2016). <https://doi.org/10.1145/2856035>
 41. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. *IEEE Softw.* **23**(2), 62–70 (2006). <https://doi.org/10.1109/MS.2006.61>
 42. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.* **19**(1), 17–37 (1982). [https://doi.org/10.1016/0004-3702\(82\)90020-0](https://doi.org/10.1016/0004-3702(82)90020-0)
 43. France, R., Rumpe, B.: Model-driven development of complex software a research roadmap. *Future Softw. Eng. FOSE'07* (2007). <https://doi.org/10.1109/FOSE.2007.14>
 44. García-Domínguez, A., Bencomo, N., Parra-Ullauri, J.M., García-Paucar, L.H.: Querying and annotating model histories with time-aware patterns. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp 194–204 (2019). <https://doi.org/10.1109/MODELS.2019.000-2>
 45. Garcia Dominguez, A., Bencomo, N., Parra Ullauri, J.M., Garcia Paucar, L.H.: towards history-aware self-adaptation with explanation capabilities. In: 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W), IEEE, Umea, Sweden, pp 18–23 (2019). <https://doi.org/10.1109/FAS-W.2019.00018>
 46. Garlan, D., Schmerl, B., Cheng, S.W.: Software architecture-based self-adaptation. In: Zhang, Y., Yang, L.T., Denko, M.K. (eds.) *Autonomic Computing and Networking*, pp. 31–55. Springer, US, Boston MA (2009)
 47. Ghahremani, S., Giese, H., Vogel, T.: Improving scalability and reward of utility-driven self-healing for large dynamic architectures. *ACM Trans. Auton. Adap. Syst.* **14**(3), 121–1241 (2020). <https://doi.org/10.1145/3380965>
 48. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. *Electron. Commun. EASST* (2009). (<https://doi.org/10.14279/tuj.eceasst.18.268>)
 49. Giese, H., Lambers, L., Becker, B., Hildebrandt, S., Neumann, S., Vogel, T., Wätzoldt, S.: Graph transformations for MDE, adaptation, and models at runtime. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) *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*, pp. 137–191. Springer, Berlin, Heidelberg (2012)
 50. Giese, H., Maximova, M., Sakizloglou, L., Schneider, S.: Metric temporal graph logic over typed attributed graphs. In: Hähnle, R., van der Aalst, W. (eds.) *Fundamental Approaches to Software Engineering*, pp. 282–298. Springer International Publishing, Cham (2019)
 51. Gómez, A., Cabot, J., Wimmer, M.: TemporalEMF: a temporal metamodeling framework. In: Trujillo, J.C., Davis, K.C., Du, X., Li, Z., Ling, T.W., Li, G., Lee, M.L. (eds.) *Conceptual Modeling*, pp. 365–381. Springer International Publishing, Cham (2018)
 52. Google Inc (n.d.) Google Core Libraries for Java. <https://github.com/google/guava>, Accessed on 23 Oct 2021
 53. Grez, A., Riveros, C., Ugarte, M., Vansummeren, S.: On the Expressiveness of Languages for Complex Event Recognition. In: Lutz, C., Jung, J.C. (eds.) *23rd International Conference on Database Theory (ICDT 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik. Dagstuhl, Germany, Leibniz International Proceedings in Informatics (LIPIcs), vol 155, pp 15:1–15:17 (2020). <https://doi.org/10.4230/LIPIcs.ICDT.2020.15>
 54. Gyapay, S., Heckel, R., Varró, D.: Graph transformation with time: causality and logical clocks. In: Goos, G., Hartmanis, J., van Leeuwen, J., Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) *Graph Transformation*, vol. 2505, pp. 120–134. Springer, Berlin Heidelberg (2002)
 55. Habel, A., Pennemann, Kh.: Correctness of high-level transformation systems relative to nested conditions. *Math. Struct. Comput. Sci.* **19**(2), 245–296 (2009)
 56. Haeusler, M., Trojer, T., Kessler, J., Farwick, M., Nowakowski, E., Breu, R.: Chronosphere: a graph-based emf model repository for IT landscape models. *Softw. Syst. Model.* **18**(6), 3487–3526 (2019). <https://doi.org/10.1007/s10270-019-00725-0>
 57. Hallé, S.: When RV meets CEP. In: *Runtime Verification*, pp. 68–91. Springer, Cham (2016)
 58. Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: A Graph Engine for Temporal Graph Analysis. In: *Proceedings of the Ninth European Conference on Computer Systems*, ACM, New York, NY, USA, EuroSys '14, pp 1:1–1:14 (2014). <https://doi.org/10.1145/2592798.2592799>
 59. Hartmann, T., Fouquet, F., Jimenez, M., Rouvoy, R., Traon, Y.L.: Analyzing Complex Data in Motion at Scale with Temporal Graphs. In: He, X. (ed.) *The 29th International Conference on Software Engineering and Knowledge Engineering*, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 5–7, 2017, KSI Research Inc. and Knowledge Systems Institute Graduate School, pp 596–601 (2017). <https://doi.org/10.18293/SEKE2017-048>
 60. Havelund, K., Peled, D.: Efficient runtime verification of first-order temporal properties. In: Merino, P. (ed.) *Gallardo MdM*, vol. 10869, pp. 26–47. Springer International Publishing, Cham (2018a)
 61. Havelund, K., Peled, D.: Runtime verification: from propositional to first-order temporal logic. In: Colombo, C., Leucker, M. (eds.) *Runtime Verification*, pp. 90–112. Springer International Publishing, Cham (2018b)
 62. Havelund, K., Peled, D.: BDDs for representing data in runtime verification. In: Deshmukh, J., Ničković, D. (eds.) *Runtime Verification*, vol. 12399, pp. 107–128. Springer International Publishing, Cham (2020a)
 63. Havelund, K., Peled, D.: First-order timed runtime verification Using BDDs. In: Hung, D.V., Sokolsky, O. (eds.) *Automated Technology for Verification and Analysis*, pp. 3–24. Springer International Publishing, Cham (2020b)
 64. Hildebrandt, S.: On the Performance and Conformance of Triple Graph Grammar Implementations. PhD thesis, Hasso Plattner Institute at the University of Potsdam (2014)
 65. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase - a deductive object base for meta data management. *J. Intell. Inf. Syst.* **4**(2), 167–192 (1995). <https://doi.org/10.1007/BF00961873>
 66. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36**(1), 41–50 (2003). <https://doi.org/10.1109/MC.2003.1160055>
 67. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp 997–1008 (2013). <https://doi.org/10.1109/ICDE.2013.6544892>
 68. Klein, F., Giese, H.: Joint structural and temporal property specification using timed story scenario diagrams. In: *International Conference on Fundamental Approaches to Software Engineering*, Springer, pp 185–199 (2007)

69. Koymans, R.: Specifying real-time properties with metric temporal logic. *Real-Time Syst.* **2**(4), 255–299 (1990). <https://doi.org/10.1007/BF01995674>
70. Krause, C., Johannsen, D., Deeb, R., Sattler, K.U., Knacker, D., Niadzelka, A.: An SQL-based query language and engine for graph pattern matching. In: Echahed, R., Minas, M. (eds.) *Graph Transformation*, vol. 9761, pp. 153–169. Springer International Publishing, Cham (2016)
71. Kumar, P., Huang, H.H.: GraphOne: a data store for real-time analytics on evolving graphs. *ACM Trans. Storage* **15**(4), 29:1–29:40 (2020). <https://doi.org/10.1145/3364180>
72. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. *Form. Methods Syst. Des.* **19**(3), 291–314 (2001). <https://doi.org/10.1023/A:1011254632723>
73. Lanese, I., Bucchiarone, A., Montesi, F.: A framework for rule-based dynamic adaptation. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) *Trustworthy Global Computing*, pp. 284–300. Springer, Berlin, Heidelberg (2010)
74. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, IEEE*, pp. 383–392 (2002)
75. LDBC Social Network Benchmark task force (2021) The LDBC Social Network Benchmark (version 0.4.0). Tech. rep., Linked Data Benchmark Council
76. Liu, L., Jeusfeld, M.A.: Suitability of active rules for model transformation. In: Kirikova, M., Stirna, J. (eds.) *Proceedings of the CAiSE'12 Forum at the 24th International Conference on Advanced Information Systems Engineering (CAiSE)*, Gdansk, Poland, June 28, 2012, CEUR-WS.org, CEUR Workshop Proceedings, vol. 855, pp. 131–138 (2012)
77. Magee, J., Kramer, J.: Dynamic structure in software architectures. *ACM SIGSOFT Softw. Eng. Notes* **21**(6), 3–14 (1996). <https://doi.org/10.1145/250707.239104>
78. Mannhardt, F., Blinde, D.: Analyzing the Trajectories of Patients with Sepsis Using Process Mining. In: Gulden, J., Nurcan, S., Reinhartz-Berger, I., Guédria, W., Bera, P., Guerreiro, S., Fellmann, M., Weidlich, M. (eds.) *RADAR+EMISA@CAiSE*, Essen, Germany, June 12–13, 2017, CEUR-WS.org, CEUR Workshop Proceedings, vol. 1859, pp. 72–80 (2017)
79. Massri, M., Raipin, P., Meyer, P.: GDBALive: a temporal graph database built on top of a columnar data store. *J. Adv. Inf. Technol.* (2021). <https://doi.org/10.12720/jait.12.3.169-178>
80. Mazak, A., Wolny, S., Gómez, A., Cabot, J., Wimmer, M., Kappel, G.: Temporal models on time series databases. *J. Object Technol.* **19**(3), 31 (2020). <https://doi.org/10.5381/jot.2020.19.3.a14>
81. n MDELab (2020) InTempo Eclipse Plugin Homepage. <https://www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/intempo/>, Accessed on 23 Oct 2021
82. Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., Chen, W.: ImmortalGraph: a system for storage and analysis of temporal graphs. *ACM Trans. Storage* **11**(3), 14:1–14:34 (2015). <https://doi.org/10.1145/2700302>
83. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to interval analysis. *Soc. Ind. Appl. Math.* (2009). <https://doi.org/10.1137/1.9780898717716>
84. Mouline, L., Benelallam, A., Fouquet, F., Bourcier, J., Barais, O.: A Temporal Model for Interactive Diagnosis of Adaptive Systems. In: *2018 IEEE International Conference on Autonomic Computing (ICAC)*, IEEE, Trento, pp. 175–180 (2018). <https://doi.org/10.1109/ICAC.2018.00029>
85. National Health Service UK (2016) Records Management Code of Practice for Health and Social Care 2016. <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>, Accessed on 23 Oct 2021
86. Office for Civil Rights (OCR) (2008) Summary of the HIPAA Privacy Rule. <https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html>, Accessed on 23 Oct 2021
87. Peled, D., Havelund, K.: Refining the safety-liveness classification of temporal properties according to monitorability. In: Margaria, T., Graf, S., Larsen, K.G. (eds.) *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, pp. 218–234. Springer International Publishing, Cham (2019)
88. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) *Graph Transformations*, pp. 319–335. Springer, Berlin, Heidelberg (2004)
89. Rhodes, A., Evans, L.E., Alhazzani, W., Levy, M.M., Antonelli, M., Ferrer, R., Kumar, A., Sevransky, J.E., Sprung, C.L., Nunnally, M.E., et al.: Surviving sepsis campaign: international guidelines for management of sepsis and septic shock: 2016. *Intensive Care Med.* **43**(3), 304–377 (2017)
90. Richters, M., Gogolla, M.: OCL: syntax, semantics, and tools. In: Clark, T., Warmer, J. (eds.) *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pp. 42–68. Springer, Berlin, Heidelberg (2002)
91. Rötschke, T., Schürr, A.: Temporal graph queries to support software evolution. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) *Graph Transformations*, pp. 291–305. Springer, Berlin, Heidelberg (2006)
92. Roy, P.C., Abidi, S.R., Abidi, S.S.R.: Monitoring medication adherence in smart environments in the context of patient self-management a knowledge-driven approach. In: *Smart Technologies in Healthcare*, pp. 195–223. CRC Press, Florida (2017)
93. Sakizloglou, L., Ghahremani, S.: Simplified Event Logs for Sepsis Patient Trajectories (2020). <https://zenodo.org/record/3989590>, Accessed on 23 Oct 2021
94. Sakizloglou, L., Ghahremani, S., Barkowsky, M., Giese, H.: A Scalable Querying Scheme for Memory-Efficient Runtime Models with History. In: Syriani, E., Sahraoui, H.A., de Lara, J., Abrahão, S. (eds.) *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*, Virtual Event, Canada, 18–23 October, 2020, ACM, pp. 175–186 (2020a). <https://doi.org/10.1145/3365438.3410961>
95. Sakizloglou, L., Ghahremani, S., Brand, T., Barkowsky, M., Giese, H.: 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*, Association for Computing Machinery, New York, NY, USA, SEAMS '20, pp. 188–194 (2020b). <https://doi.org/10.1145/3387939.3388614>
96. Sakizloglou, L., Ghahremani, S., Barkowsky, M.: Event Logs Containing Deletions (2021). <https://doi.org/10.5281/zenodo.5042439>, Accessed on 23 Oct 2021
97. Sánchez Cuadrado, J., de Lara, J.: Streaming model transformations: scenarios, challenges and initial solutions. In: Duddy, K., Kappel, G. (eds.) *Theory and Practice of Model Transformations*, pp. 1–16. Springer, Berlin, Heidelberg (2013)
98. SAP OrientDB (n.d.) Live Query · OrientDB Manual. <http://orientdb.com/docs/3.1.x/java/Live-Query.html>, Accessed on 23 Oct 2021
99. Schneider, S., Lambers, L., Orejas, F.: Automated reasoning for attributed graph properties. *Int. J. Softw. Tools Technol. Transf.* **20**(6), 705–737 (2018). <https://doi.org/10.1007/s10009-018-0496-3>
100. Schneider, S., Sakizloglou, L., Maximova, M., Giese, H.: Optimistic and pessimistic on-the-fly analysis for metric temporal graph logic. In: Gadducci, F., Kehrer, T. (eds.) *Graph Transformations*

- mation, pp. 276–294. Springer International Publishing, Cham (2020)
101. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
 102. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: a distributed incremental model query framework in the cloud. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) *Model-Driven Engineering Languages and Systems*, pp. 653–669. Springer International Publishing, Cham (2014)
 103. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Softw. Syst. Model.* **15**(1), 31–69 (2016). <https://doi.org/10.1007/s10270-013-0394-9>
 104. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries. *Sci. Comput. Program.* **98**, 80–99 (2015). <https://doi.org/10.1016/j.scico.2014.01.004>
 105. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Softw. Syst. Model.* **15**(3), 609–629 (2016). <https://doi.org/10.1007/s10270-016-0530-4>
 106. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Association for Computing Machinery, New York, NY, USA, SEAMS '10, pp. 39–48 (2010). <https://doi.org/10.1145/1808984.1808989>
 107. Vogel, T., Seibel, A., Giese, H.: The role of models and megamodels at runtime. In: Dingel, J., Solberg, A. (eds.) *Models in Software Engineering*, pp. 224–238. Springer, Berlin, Heidelberg (2011)
 108. Waudby, J., Steer, B.A., Prat-Pérez, A., Szárnyas, G.: 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)*, ACM, Portland OR USA, pp. 1–8 (2020). <https://doi.org/10.1145/3398682.3399165>
 109. Weyns, D., Calinescu, R.: Tele Assistance: A Self-Adaptive Service-Based System Exemplar. In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 88–92 (2015). <https://doi.org/10.1109/SEAMS.2015.27>
 110. Wilson, W., Taubert, K.A., Gewitz, M., Lockhart, P.B., Baddour, L.M., Levison, M., Bolger, A., Cabell, C.H., Takahashi, M., Baltimore, R.S., et al.: Prevention of infective endocarditis: a guideline from the american heart association. *Circulation* **116**(15), 1736–1754 (2007)
 111. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering*. Springer-Verlag, Berlin Heidelberg (2012)
 112. Zündorf, A.: *Rigorous Object Oriented Software Development*. Habilitation Thesis, University of Paderborn (2001)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Lucas Sakizloglou is a doctoral researcher at the System Analysis and Modeling group at the Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam. He is also a research and teaching fellow at the Software Systems Engineering group at Brandenburg University of Technology. His research interests include history-aware runtime models and the monitoring of temporal specifications.

Sona Ghahremani is a doctoral researcher at the System Analysis and Modeling group at the Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam. Her research interests include AI- and model-driven design and control of autonomous and self-aware computing systems.

Matthias Barkowsky is a doctoral researcher at the System Analysis and Modeling group at the Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam. His main research interest is the incremental execution of queries over evolving host graphs.

Holger Giese is a full professor at the Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam. Prior to that, he was assistant professor for object-oriented specification of distributed systems in the Software Engineering Group of the University of Paderborn. He studied Technical Computer Science at the University of Siegen and received his engineering degree in October 1995. He received a doctorate in Computer Science at the Institute of Computer Science

at the University of Münster in February 2001. His research focus is the model-driven development of software-intensive systems covering the specification of dynamic and flexible systems by services, collaborations, patterns, and components, approaches to analyze and formally verify such models, and approaches for model synthesis. The main focus are systems that are typically distributed systems, embedded real-time systems as well as systems that are capable to adapt and coordinate themselves. Furthermore, he does research on model transformation, concepts for generating source code for structure and behavior, and the general problem of model integration during the process of model-driven development. He is a member of the Association for Computing Machinery, the IEEE Computer Society, and the German Informatics Society.