Querying and annotating model histories with time-aware patterns

Antonio García-Domínguez
SEA, SARI
Aston University
Birmingham, UK
a.garcia-dominguez@aston.ac.uk

Nelly Bencomo
SEA, SARI
Aston University
Birmingham, UK
n.bencomo@aston.ac.uk

Juan Marcelo Parra-Ullauri
SEA, SARI
Aston University
Birmingham, UK
j.parra-ullauri@aston.ac.uk

Luis Hernán García-Paucar
SEA, SARI
Aston University
Birmingham, UK
garciapl@aston.ac.uk

***Abstract*—Models are not static entities: they evolve over time due to changes. Changes may inadvertently and surprisingly violate constraints imposed. Therefore, the models need to be monitored for compliance. On the one hand, in traditional design-time applications, new and evolving requirements impose changes on a model over time. These changes may accidentally break design rules. Further, the growing complexity of the models may need to be tracked for manageability. On the other hand, newer applications use models at runtime; building runtime abstractions that are used to control a system. Adopters of these approaches will need to query the history of the system to check if the models evolved as expected, or to find out the reasons for a particular behavior. Changes over models at runtime are more frequent than changes over design models. To cover these demands, we argue that a flexible and scalable approach for querying the history of the models is needed to study the evolution and for compliance sake. This paper presents a set of extensions to a model query language inspired in the Object Constraint Language (the Epsilon Object Language) for traversing the history of a model, and for making temporal assertions that will allow the elicitation of historic information. As querying long histories may be costly, the paper presents an approach that annotates versions of interest as they are observed, in order to provide efficient recalls in possible future queries. The approach has been implemented in a model indexing tool, and is demonstrated through a case study from the autonomous and self-adaptive systems domain.**

***Index Terms*—Model querying, model versioning, temporal graph databases, model indexing, scalable model-driven engineering.**

I. Introduction

Tool support exists that provides access to up-to-date versions of models. However, the corresponding tool support to study the history of models is rather limited [1]–[3]. There are multiple domains of interest on evolving models, such as evolution of design models usually modified by humans [4], [5], and runtime models modified by systems automatically [6].

Changes of design-time models take place as they are linked either to complex collaborative, co-engineering processes, or they may need to be linked to runtime concerns. While approaches based on runtime models have focused on providing an infrastructure to instantiate and manipulate runtime models, issues related to storing the history of those changes have been neglected [6]. Changes on runtime models exacerbate the challenges as the changes are more frequent.

Given a time period, changes in models require looking for a model version stored in a model repository, to retrieve data associated with the context of the changes at that time. The aims can be multiple, e.g. to check if the models evolved as expected, to support accountability, or to find out the reasons for a particular behavior in the case of self-adaptation [3], [7]. Moreover, large volumes of data coming from different sources increasingly produce historical data, creating the need for analyzing the history of running systems, discovering temporal patterns, and predicting future trends [2].

The above imply that there is increasing need for tool support to the evolution of the models, to find and study situations of interest in the past, or check if initial assumptions still apply over time and therefore, avoid surprises [8], [9]. Further, infrastructures for querying the history of models should be flexible and scalable. To cover these demands, this paper presents a set of extensions to a model query language inspired in the OMG Object Constraint Language (OCL [10]) (the Epsilon Object Language or EOL [11]) for traversing the history of a model and for making temporal assertions that will allow the elicitation of historic information. The extensions are designed to be composable so that more complex patterns (e.g. repeated intervals of varying lengths) can be expressed. As querying long histories may be costly, the presented approach can annotate versions of interest as they are observed to provide efficient recall in later queries. The approach has been implemented using a model indexing tool, and is demonstrated through a case study from the self-adaptive systems domain.

The paper is organised as follows. Section II presents the past work on time-awareness in model-driven engineering and model versioning that underlies the proposal in this paper. Section III describes our proposed model query language extensions. Section IV presents a case study with several versions of a time-aware query, evaluating the expressiveness of the extensions and their performance. Section V concludes the paper and outlines future research opportunities.

II. Related Work

This section introduces several concepts and prior work that is needed to place the paper's contributions into context. A short discussion on the integration of time into model-driven

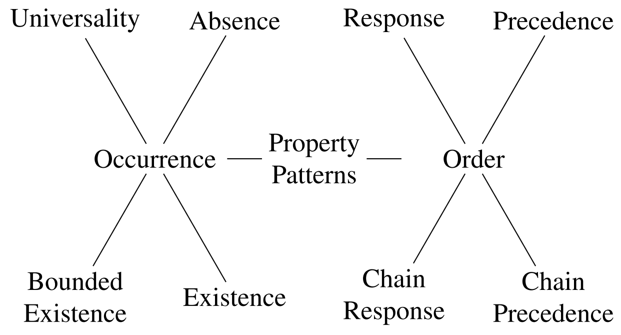


Figure 1: Hierarchy of property patterns by Dwyer [12].

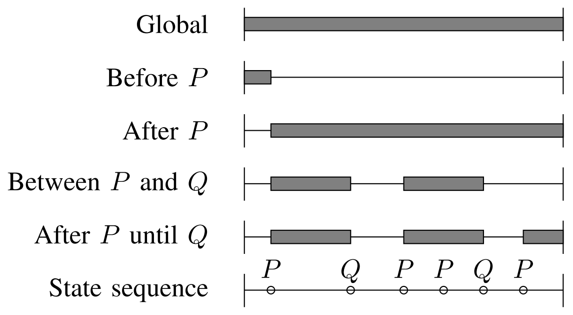


Figure 2: Pattern scopes by Dwyer [12].

engineering is followed by an overview of the solutions for storing models that evolve over time.

A. Time in model-driven engineering

There are several ways to consider time in model-driven engineering. One way is to include it in the concepts of the modelling language itself. For instance, the OMG MARTE (Modeling and Analysis of Real-Time and Embedded Systems) profile [13] includes a framework for representing time, supporting several abstractions: a causal one (modelling event precedence), a clock-based one (dividing time into "instants" in which several processes may run simultaneously), or a physical one (with real-world duration values). Different from [13], the concern of this paper is more about the evolution of the models themselves over time, along with the modifications made by either human modellers, by an automated process (e.g. the operation or monitoring process of a system) or the running system itself. As an example of early work, Ziemann and Gogolla proposed in 2002 TOCL (Temporal OCL), an extension of the OCL for describing temporal constraints on an object-oriented model [14]. Their proposal extended OCL with a number of *past expressions* (e.g. "prev e" is true if e is true in the "previous state") and *future expressions* (e.g. "always e_1 until e_2" is true if e_1 is true "from then on" until e_2 is true "for the first time in the future"). While the proposal included formal semantics and a type system, it did not include any suggestions on how to implement this language.

In 2014, later work by Kanso and Taha [15] stated that existing works on temporal OCL extensions at the time only brought syntactic proposals without any concrete implementation, and required knowledge of specific temporal logics. The work proposed a temporal extension of OCL based on the work of Dwyer, Avrunin and Corbett on specification patterns for finite-state verification [12] to make temporal logics more accessible. Dwyer et al. organised the patterns into a hierarchy (as shown in Figure 1). They also added a mapping from each pattern to various formalisms (including linear temporal logic). Dwyer et al. collected 555 specifications from at least 35 different sources, and identified the described patterns to cover 92% of them; with Response, Universality and Absence being the most common out of all three. In terms of scopes, Dwyer et al. defined five, which are shown in Figure 2: "global" (the entire execution of the program), "before" a certain event, "after" an event, "between" two events, and "after" a certain event "until" another event.

In the work by Kanso and Taha cited above, events are of two types: either a certain operation op has been invoked in a context where pre is a pre-condition and $post$ is a post-condition ("isCalled(op, pre, $post$)"), or a certain predicate has become true after some invocation ("becomesTrue(P)", a shorthand for "isCalled(any,$\neg P$,P)"). The notation was implemented as an extension of the Eclipse MDT OCL plugin, with a new grammar, new editors, and new pivot metamodels. A transformation from OCL temporal properties to test scenarios for object-oriented software was implemented. The work by Kanso and Taha focused on generating test cases, and did not consider the use of these extensions for querying the histories of the models themselves.

In 2016, Hilken and Gogolla presented a mapping of several Linear Temporal Logic primitives to standard OCL, using so-called *filmstrip models* [16]. These models conform to an enriched version of their original metamodels, which consists of a trace of model states that are linked by operation calls. This enriched structure allows for implementation of LTL operators such as "finally" or "globally" with regular OCL closures and logic quantifiers. Unfortunately, in their work the authors did not address how the filmstrip-based temporal constraints would scale to longer traces, neither in terms of both memory requirements nor processing time.

Later in 2017, Dou, Bianculli and Briand presented TemPsy [17], a pattern-based specification language inspired as well by the primtives of Dwyer et al. The TemPsy patterns were translated to OCL constraints on the fly, as requested by their industrial pattern. The constraints ran on models which conformed to dedicated trace metamodels. The authors compared their TemPsy-Check tool against MonPoly, an existing tool based on MFOTL (metric first-order temporal logic), using a case study from the eGovernment domain. TemPsy-Check had similar or better performance, while having a notation that was easier to use than MFOTL.

Beyond temporal extensions to constraint languages, Benellalam et al. argued for the need to raise time-awareness in model-driven engineering in 2017 [18], identifying gaps in the capabilities of conceptual modelling approaches, query languages and storage solutions. Their work used an example based on the smart grids domain, where a SmartMeter metaclass is annotated with time sensitivity, periodicity and precision concerns. The authors suggest that time should be a first-class property orthogonal to all model elements, allowing each element to evolve independently over time (e.g. more time-sensitive concepts would store more versions).

Mouline et al. presented a metamodel for interactive diagnosis of adaptive systems which combines design-time and run-time concerns [7]. The design-time parts cover the available strategies and actions, whereas the run-time parts cover the observations made and the decisions that were taken. The authors propose allowing users to use temporal queries to find out why a specific action was taken, helping with the self-explainability of the system.

Meyers et al. described ProMoBox in [19], a framework for building behavioural domain-specific modeling languages with the ability to define and verify temporal properties. A DSML description is turned into five DSMLs: design-time concerns, run-time concerns, event inputs, traces, and a Dwyer-inspired temporal property language. Properties are translated into Promela models for the Spin model checker.

B. Storage of evolving models

Storage solutions for evolving models have traditionally been divided into two types: file-based approaches (combined with a traditional version control system, such as Git [20]), or dedicated versioned model repositories such as the Eclipse Connected Data Objects (CDO) project [21]. File-based approaches benefit from using VCS tools that are familiar to most developers, unlike the dedicated *model repositories* which have a considerable learning curve. Additional capabilities can be added on top of the traditional VCS: for instance, Debreceni et al. used live transformations acting as security lenses on top of SVN-versioned models to implement fine-grained access control and property-level locking [22].

Traditional VCS by themselves do not provide any support for scalable querying: in the case of large models, the "simple" approach of loading all model files into memory may be unfeasible. One approach to solve this is to use a separate *model indexer* to mirror a model into a database that is amenable for querying, while allowing users to keep their VCS-based approaches. Barmpis et al. presented Hawk in [23], a system that mirrors a collection of models into a Neo4j [24] NoSQL graph database and run model queries on it. Hegedüs et al. presented IncQuery Server for Teamwork Cloud [25], a dedicated solution for querying Teamwork Cloud model repositories, with the ability to access version snapshots and querying the latest state. István et al. showed in [26] a different approach based on streaming transformations, which react to live changes in the model and are based on complex event processing and event-driven transformation concepts.

In general, NoSQL databases have been seen as a promising solution for the future needs of storing very large object-oriented models. The NeoEMF project [27] has implemented several alternative storage layers for models implemented in the Eclipse Modeling Framework (EMF). Each of these layers has a different focus: optimizing queries, accelerating updates, or scaling in the cloud, respectively.

NoSQL databases are also being extended to cover the time domain. Moffit and Stoyanovich presented a formal extension of the property label graph model called *TGraph* [28], with support for edge and node creation. Hartmann presented another formal and practical definition of *temporal graphs* [29], and demonstrated GreyCat, a temporal graph database (TGDB) management system which implemented the idea. GreyCat uses a compact representation based on *state chunks*, which only stores nodes that have changed between versions. Since the time they published their work, other temporal graph databases have been proposed, such as ChronoGraph [30].

One disadvantage of GreyCat is that it is not EMF-compliant, making it incompatible with a number of existing tools. An alternative called TemporalEMF was presented by Gómez et al., a followup to the above vision paper from Benellalam [1]. TemporalEMF is a temporal meta-modelling framework which extends EMF with a dedicated profile for annotating types and operations with temporal concerns. For instance, types can be marked as requiring temporal support (with explicit durability / storage frequency), and operations can be marked to be logged or to erase ("vacuum") past history. TemporalEMF has a dedicated model storage layer on top of HBase [31], and some limited support for querying these temporal objects: an *eGetAt(instant, feature)* operation which produces the value of the feature at that instant, and an *eGetAllBetween(start, end, feature)* which produces all the values within a time range. Unfortunately, it can be argued that with only these two operations, it would not be possible to implement the scopes and primitives identified by Dwyer (Figures 1 and 2). For instance, the Universality pattern would require a way to retrieve all distinct versions of a model element. We argue that users wishing to retain their existing model stores would not be able to use TemporalEMF, either.

III. Model History Querying and Annotation

This section proposes a set of extensions on query languages for making temporal assertions about the history of a model. An optimisation that pre-computes versions of interest for the query is also presented.

A. Time-aware patterns

Traditional model querying languages have focused on extracting information from a single version of the model. Languages like the OMG Object Constraint Language [10] or the Epsilon Object Language [11] have primitives for quickly finding all objects of a type (`Type.allInstances`) and for finding elements that meet certain predicates. However, they do not expose the history of the model; in other words, it is not possible, for instance, to ask the model for the element which "had no children in the previous version" or the element which "has always had a value over a threshold". Computing this information with the current available languages would essentially require *ad hoc* implementations that would both repeat a query over the history and add specific logic over these repeated queries.

Operation	Syntax
VERSION TRAVERSAL	
All versions, from newest to oldest	`x.versions`
Versions within a range	`x.getVersionsBetween(from, to)`
Versions from a time point (included)	`x.getVersionsFrom(from)`
Versions up to a time point (included)	`x.getVersionsUpTo(to)`
Earliest / latest version	`x.earliest`, `x.latest`
Next / previous version	`x.next`, `x.prev` / `x.previous`
Version timepoint	`x.time`
TEMPORAL ASSERTIONS	
True for all versions?	`x.always(v | p)`
False for all versions?	`x.never(v | p)`
Any matching version?	`x.eventually(v | p)`
At least n matching?	`x.eventuallyAtLeast(v | p, n)`
At most n matching?	`x.eventuallyAtMost(v | p, n)`
PREDICATE-BASED VERSION SCOPING	
View with versions since match (inclusive, exclusive)	`x.since(v | p)`, `x.after(v | p)`
... until match (i., e.)	`x.until(v | p)`, `x.before(v | p)`
... with matching	`x.when(v | p)`
CONTEXT-BASED VERSION SCOPING	
View with versions since current (i., e.)	`x.sinceThen`, `x.afterThen`
... until current (i., e.)	`x.untilThen`, `x.beforeThen`
VERSION UNSCOPING	
Original without version scoping	`x.unscoped`

Table I: Proposed new operations for time-awareness for the Epsilon Object Language, divided by type. `p` stands for a Boolean predicate.

However, as it is possible to keep models in versioned stores (e.g. CDO [21]) or temporal graph databases (e.g. Greycat [29]), as discussed in Section II-B, it should be feasible to integrate the traversal of the history into the querying languages themselves. Additional primitives would allow users to visit previous versions of a model element in the same query, or to find out the instances of a type that were available at some earlier point in time.

To find model elements according to these time-aware patterns, users need a conceptual model about what "the versions of `x`" means. In these queries, `x` may be a model element (in the Eclipse Modeling Framework, an EOBJECT), or it may be a type of model element (in EMF, an ECLASS). `x` has a certain identity and a lifespan:

- Model elements may receive an identity through a natural identifier (a "business key"), an artificial identifier (e.g. XMI identifiers), or their location within the model. Their lifespans are limited by the moment they are created, and when they are destroyed: even if an object with the same identity and/or state was recreated later, it would be treated as a different lifespan. Model elements have a new version every time one of their features change.
- Types are identified by a combination of the metamodel identifier (in EMF, the metamodel URL) and its name. Types are "immortal", as their lifespans start at the "beginning of time" and end at the "end of time" (the latest timepoint that we may be able to represent). The initial version of a type has no instances. A type will have a new version when instances are created or destroyed.

With the lifespans and the versions of both model elements and their types, time-awareness can be added as a new set of operations. These operations are collected in Table I, as formulated for the Epsilon Object Language, which is inspired by OCL. The operations can be divided into several types, which will be discussed in separate subheadings below.

1) Basic version traversal: These primitives provide access to all the versions of a type or a model element. To retrieve the instances of the latest version of a type, a user would write `Type.latest.all` instead of `Type.all`. The new semantics of `Type.all` would need to default to either the earliest or the latest version: in this paper, the default is the earliest version of the type. `x.versions` would return the collection of the versions of `x`: therefore, finding the versions of `x` that meet a predicate could be done with `x.versions.select(version | predicate)`. `Type.latest.all.first` would produce the first instance of that type in its latest version: the instance would be at the timepoint of the latest version of that type.

2) Temporal assertions: These operations provide concise ways to check if certain temporal properties hold for a type or model element. For instance, `x.always(version | predicate)` would answer if a particular predicate has always held for `x`. This could already be done with `x.versions.forAll(version | predicate)`. However, it is easier to read and it is more amenable for static analysis in the future (e.g. for turning time-aware patterns into temporal constraints). `eventuallyAtLeast` and `eventuallyAtMost` would not be as easy to implement in one line, given their ability to stop as soon as enough or too many elements are found, respectively.

3) Predicate-based version scoping: Sometimes, we may want to examine specific version ranges, e.g. from the first moment P was true onwards. Rather than extending the assertions with arguments for version ranges, a better design is to have dedicated primitives which will return a view of the original type or model element, which only exposes some of the versions. This can produce richer patterns by composition: for example, with `x.since(v | Q).always(v | R)`, we would be able to check whether since an arbitrary predicate `Q` first held from the current timepoint of `x`, `R` was always true. The scoping can also be composed: for instance, `x.since(v|Q).until(v|R)` would produce a view over `x` that would only report the versions between `Q` and `R`. The returned views are still model element or types, as the originals, and the view will always switch to the oldest matching version within the scope. For instance, if `x` was a PERSON, `x.since(v|Q)` will still be the same PERSON, but at the first timepoint when `Q` held. In these operations, if no

```
s.when(v | v.color = 'yellow'
  and (v.prev.isUndefined()
   or v.prev.color <> 'yellow'))
 .always(v | v.unscoped.sinceThen
  .before(v | v.color <> 'yellow')
   .versions.collect(v | v.count).sum() <= 5)
```

Listing 1: Example Epsilon Object Language query for the `unscoped` primitive: SEMAPHORE `s` does not let more than 5 vehicles through in any of its yellow intervals.

matching version exists, an undefined value will be returned (e.g. `oclUndefined` in OCL or `null` in other languages).

4) Context-based version scoping: As said above, every type and model element visited at each step of a query will be at a specific timepoint. This timepoint may be used as the reference for the version scoping: there are `xThen` variants of `since`, `after`, `before` and `then` that use the timepoint of the current model element or type rather than a predicate.

5) Version unscoping: Crucially, one final operation which was not part of the works by Dwyer and Kanso is the `unscoped` operation. This operations returns the same model element or type, but without any of the version scoping: all versions are visible once more. This is useful in more advanced queries which need to switch between different scopes in the same query. As an example, suppose that we have a SEMAPHORE model element `s` with two features: the shown `color` (red, yellow or green), and the `count` with the number of vehicles that passed in the last 5 seconds. Assume that we want to check, for example, if across all intervals where the light was yellow, no more than 5 vehicles passed.

In terms of the querying primitives, we would first need to find the versions that delimit these intervals, i.e. when the light changes to yellow: either there was no previous version, or the previous version had a color other than yellow. We would normally want to test `.always(v | predicate)` on each of those delimiters, but the predicate would be evaluated on a view `v` which would only expose the versions when the semaphore changed to yellow. In order to access the right interval from within the `always` predicate, it would be necessary to undo the scoping first with `unscoped`. The complete query would be as in Listing 1: within `always`, the scoping is undone, and the vehicle counts for all versions from that point until the semaphore changes color are added together and compared with the upper bound.

B. From patterns and scopes to primitives

As discussed in Section II-A, Dwyer studied 555 specifications of temporal behaviour in various systems, and defined a set of patterns and scopes that covered 92% of them (Figures 1 and 2). These patterns and scopes were designed to be easier to understand than formalisms such as linear temporal logic. Accordingly, this section will examine the level of coverage of the patterns and scopes achieved by the primitives in Table I.

The version scoping primitives are a close match to the Occurrence subtree of Dwyer's patterns. Universality is implemented by `always`, Absence by `never`,

```
s.whenAnnotated('becomesYellow')
 .always(v | v.unscoped.sinceThen
   .before(v | v.color <> 'yellow')
   .versions.collect(v | v.count).sum() <= 5)
```

Listing 2: Optimised version of Listing 1 using timeline annotation.

Existence by `eventually`, and Bounded Existence by `eventuallyAtLeast` and `eventuallyAtMost`.

The Order primitives can be implemented by combining the available primitives, as shown in Table II:

- Precedence (P always preceded by Q) is done by finding when P occurs, and then checking that in these timepoints, Q happens at some point before P. The example uses `untilThen` to allow for P and Q to happen in the same timepoint: replacing it with `beforeThen` would require Q to happen strictly before P.
- Response (P always followed by Q) is similar, but in this case we use `sinceThen` to check that Q happens in the same or a later timepoint than P.
- Chain Precedence is a generalisation of Precedence, which deals with event chains rather than individual events. The implementation works by first finding the moment when P_1 holds and is followed by $P_2, P_3, \ldots, P_n$. For each of those points in time, we check that before that moment Q_m happens, and before that moment Q_{m-1}, and so on until Q_1. Ideally, it would be better to have primitives which take not just one predicate, but a sequence of predicates. In EOL, this could take the form of `whenChain((v1|P1), (v2|P2), ..., (vN|PN))`, for instance: the additional brackets are needed by its grammar.
- Chain Response is similar to Chain Precedence, but in this case we look first for the versions when P_n happened and was preceded by $P_{n-1}, P_{n-2}, \ldots, P_1$. We check next if for all those versions, it is always true that eventually Q_1 happens after P_n, then Q_2, and so on until Q_m.

The scopes defined by Dwyer can also be mapped to the primitives, as shown in Table III. Global is the default scope. Before is implemented by `until` (matching timepoint is included) and `before` (matching timepoint is excluded). After is implemented by `since` and `after`.

Between P and Q requires combining several primitives: `when` is used to locate occurrences of P, and then the temporal assertion of interest (e.g. `always`) will be used to check if a certain predicate holds for the intervals that start at each of those occurrences and end at the next occurrence of Q. This is another candidate for future improvement, with a simpler `between((v|P), (w|Q))` primitive.

After P until Q is similar, but the end of time also counts as a valid end of an interval. This can be detected by checking if the `next` version is undefined. This code could be replaced by a dedicated `afterUntil((v|P), (w|Q))` primitive.

Pattern	Meaning	Implementation
Precedence	P always preceded by Q	`x.when(v|P).always(v|v.unscoped.untilThen.eventually(v|Q))`
Response	P always followed by Q	`x.when(v|P).always(v|v.unscoped.sinceThen.eventually(v|Q))`
Chain Precedence	$P_1 \dots P_n$ always preceded by $Q_1 \dots Q_m$	`x.when(v	` $P_1$ **`and`** `v.sinceThen.eventually(`   `v	` $P_2$ **`and`** `v.sinceThen.eventually(...)` `).always(v	v.unscoped.untilThen.eventually(`   `v	` $Q_m$ **`and`** `v.untilThen.eventually(`     `v	` $Q_{m-1}$ **`and`** `v.untilThen.eventually(...)))`
Chain Response	$P_1 \dots P_n$ always followed by $R_1 \dots R_m$	`x.when(v	` $P_n$ **`and`** `v.untilThen.eventually(`   `v	` $P_{n-1}$ **`and`** `v.untilThen.eventually(...)` `).always(v	v.unscoped.sinceThen.eventually(`   `v	` $Q_1$ **`and`** `v.sinceThen.eventually(`     `v	` $Q_2$ **`and`** `v.sinceThen.eventually(...)))`

Table II: Dwyer’s Order patterns using the primitives from Table I.

Scope	Implementation
Global	(default: no code needed)
After P	`x.since(v|`P`)` (inclusive), `x.after(v|`P`)` (exclusive)
Before P	`x.until(v|`P`)` (inclusive), `x.before(v|`P`)` (exclusive)

Between P and Q	`x.when(v|`P`)`
	 `.temporalAssertionOp(v |`
	  `v.unscoped.sinceThen`
	   `.until(v|`Q`).predicate())`

After P until Q	`x.when(v|`P`)`
	 `.temporalAssertionOp(v |`
	  `v.unscoped.sinceThen.until(v|`
	   Q **`or not`** `v.next.isDefined())`
	   `.predicate())`

Table III: Dwyer’s scopes using the primitives from Table I.

C. *Timeline annotation*

Scalability is a challenge when indexing models with potentially very long histories: both in terms of storage size, and in terms of query times. Finding rare events across many versions can take a long time if we need to iterate through each versions while testing a predicate. For design-time models, this may happen in artefacts that have been worked on for a long time, whereas runtime models may quickly run into this problem if they are updated very frequently. Being able to answer a query as soon as possible could be critical for approaches based on runtime models which need to interact with the outside world.

This challenge is similar to the scenario when subsets of model elements that match certain predicates need to be found in very large models. Barmpis et al. proposed computing these predicates in advance, storing them into *derived attributes* that extended existing types, and keeping the model elements indexed by these values [32]. Finding which elements matched a predicate could be done with an efficient lookup, rather than expensive iteration. The same work introduced an incremental approach for maintaining such an index up to date, which minimized the number of re-evaluations of the derived attributes. This approach worked by tracking the elements and properties accessed during the evaluation of each derived attribute, and only re-evaluating the attribute when needed.

The same approach can be adopted for finding occurrences of an event in the history of a model instance. The predicate describing such an event can be defined as a derived Boolean attribute on its type, which would be pre-computed and indexed incrementally across the versions of each of its instances. The index will need to be extended so that all versions of the derived attribute are tracked, and not just the most recent one. Queries will naturally be multidimensional in this case: they will have to look up matches based on the desired value, and the appropriate range of timepoints.

In effect, such an extension is annotating the timeline of each model element with markers for these events of interest. These markers can be used in queries through alternative versions of the version scoping primitives of Table I. `whenAnnotated(a)` will produce a view that exposes the versions for which the derived Boolean attribute named `a` will be true, by index lookup. `sinceAnnotated(a)` / `afterAnnotated(a)` will produce views with left-closed and left-open version ranges by index lookup. `untilAnnotated(a)` / `beforeAnnotated(a)` will produce views with right-closed and right-open version ranges.

Returning to the SEMAPHORE example, we may be interested in precomputing when the light became yellow. We can extend the SEMAPHORE type with a derived `becomesYellow` Boolean attribute, defined with the EOL expression below. `self` is a variable set to each of the SEMAPHORE instances during evaluation:

```
return self.color = 'yellow'
  and (self.prev.isUndefined()
       or self.prev.color <> 'yellow');
```

Having pre-computed these events, the query in Listing 1 can be simplified to that of Listing 2. Rather than having to visit the entire history of the SEMAPHORE `s`, the events of interest can be simply looked up more efficiently. Accordingly, it would be possible to annotate the points in time when the SEMAPHORE stops being yellow as a derived attribute as well: users would decide which situations are worth indexing.

Beyond model queries, these annotations could also be used for incrementally computed model monitoring as well. If a certain model element matched a certain predicate, it could trigger a notification to an external system or a modeller. Users would be able to quickly recall past occurrences of the same event, for the sake of comparison. ’;.,

IV. Case Study

This section applies an implementation of the primitives to a case study from the domain of self-adaptive systems (SAS) using models@run.time [6], [33]: a Remote Data Mirroring (RDM) system [34], [35]. A self-adaptive system frequently updates its knowledge about the environment and its decision processes. Being able to query this evolution would allow users to know the reasons why particular actions were taken.

The section starts with a description of the RDM system, and then lists several research questions on the impact and trade-offs of the new primitives. This is followed by an outline of the experimental setup, and a discussion of the results.

A. System under study: RDM

The RDM system (henceforth RDM) is composed of data servers and network links. It must replicate and distribute data in an efficient manner by minimizing consumed bandwidth and ensuring the distributed data is not lost or corrupted [34], [35]. RDM can be configured by using two different topologies: minimum spanning tree (MST) and redundant topology (RT). The system selectively activates and deactivates network links to change its topology at runtime [36]. Each topology has different tradeoffs between reliability, performance and cost.

The RDM network has been modeled and implemented as a SAS that evolves over time slices. At each time slice, RDM estimates a reward level for the available adaptation actions (using MST or RT), by considering the current levels of satisficement of the non-functional requirements (NFRs) and estimating their future levels of satisficement. RDM then selects the action with the highest reward.

Three NFRs have been taken into account: Maximization of Reliability (MR), Maximization of Performance (MP) and Minimization of Cost (MC). Whether the NFRs are met or not is not directly observable (i.e. monitored). Instead, based on Bayesian inference, RDM estimates their levels of satisficement using three monitoring (MON) variables: Range of Bandwidth Consumption (RBC), Active Network Links (ANL), and Total Time for Writing (TTW). The MC and MP NFRs have higher satisficement with lower values of RBC and TTW. The MR NFR favors high values of ANL.

The work presented aims to support runtime queries about the history of the system to get more insights about the decisions made (e.g. impact on performance), or to find out the reasons for a particular surprising behavior. To assist in querying its decision processes, an abstraction of the internal state of RDM at each timeslice is saved from timeslice to timeslice into a model conforming to the metamodel in Figure 3. This abstraction includes the DECISION to be made, the available adaptation ACTIONs, the OBSERVATIONs of the environment, its beliefs on the NFRs (i.e. probabilities about the satisficement levels), and the REWARDTABLE which values the alternatives.

B. Research questions

RDM produces a sequence of trace models amenable for temporal queries, but it is yet to be seen how useful and efficient these queries would be. Two research questions were targeted by the case study:

- RQ1: how effective are the temporal assertions and version scoping primitives in Table I in making time-aware patterns more concise and less error-prone to write? Section III-B suggests that the full set can express Dwyer's patterns and scopes, but the basic version traversal primitives may be enough.
- RQ2: what are the overheads of adding time-awareness to a model-based solution, with and without timeline annotation? Timeline annotation is additional processing in exchange for fast lookup of rare events. If the tradeoff is worth it, we should be saving more time in the queries than the additional time spent on annotating.

C. Experimental setup

In order to answer RQ1 and RQ2, it was necessary to implement a storage solution which would keep track of all the versions of the trace model conforming to Figure 3, and implement the new primitives in Table I so they integrated with the storage solution.

Given the complexity of RDM, it was decided that rather than replacing its storage approach entirely, an external system would be used to index the trace models into a GreyCat temporal graph database [29]. The system implemented the approaches described by Barmpis et al. for indexing [23] and incremental derived attribute computation [32], and used EOL as its query language. The external system was told over a web service API about each new version of the trace model. The API could be used to answer queries as well.

The primitives were implemented by adding a new *operation factory* to the EOL runtime environment. This concept allows new operations to be added to existing types without modifications to its parser, unlike the temporal OCL approaches in Section II-A which required invasive changes in the grammar. The operations take advantage of the GreyCat graph APIs for traversing the history of each node in the graph.

The resulting combination of RDM and the indexing system was invoked over pseudorandom simulations of 1000 timeslices, with the same fixed seed. To simulate live querying from users and estimate the increase in query times over longer histories (for RQ2), each run invoked between timeslices one of the queries developed below.

The experiments were run on a Lenovo Thinkpad T480 with an Intel i7-8550U CPU that operates at 1.80GHz, running Ubuntu 18.04.2 LTS with Oracle Java version 1.8.0_201 and memory 15.6 GB, HDD.

D. Query selection to get further insights of runtime behaviour

Before the case study presented in this paper, the developers of RDM had experimented with the basic version of traversal primitives of Table I (`next`, `prev` and others). These primitives have proved to be useful to extract information to study and get more insights about the performance of the system. See the performance graphs in Figure 4a. For instance, the graphs show how RDM was operating within SLA agreements,

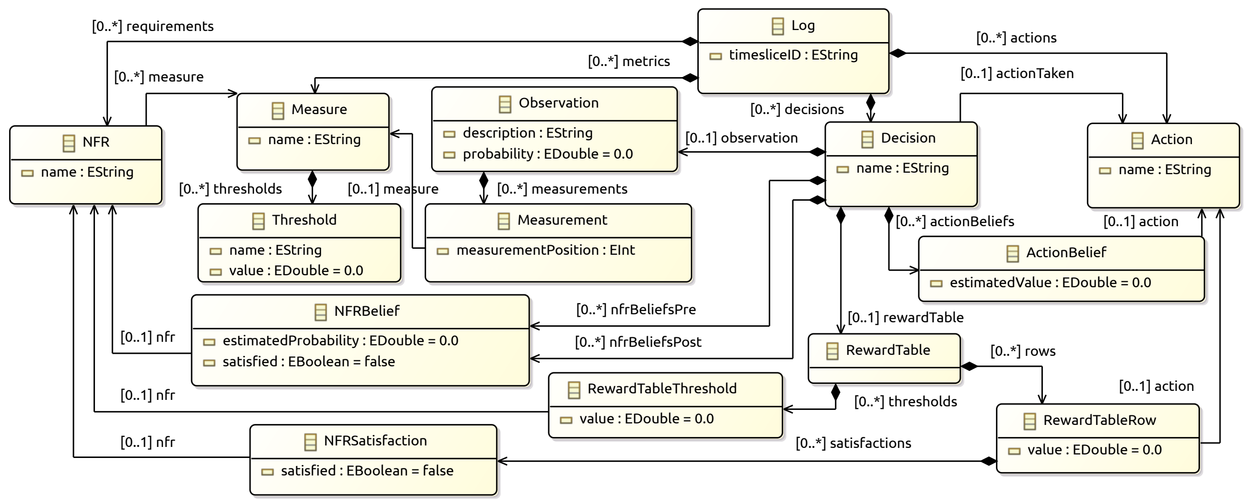
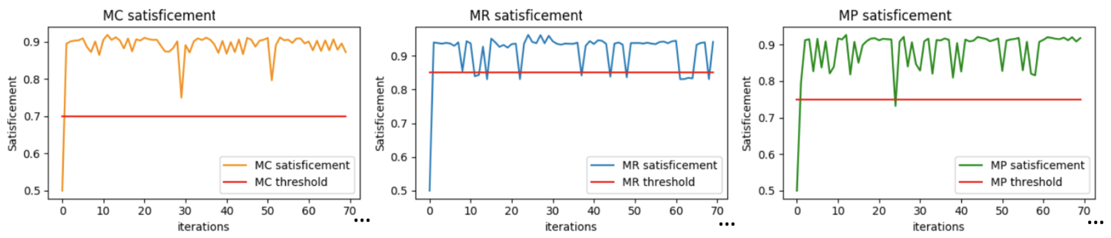
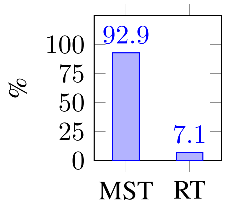


Figure 3: Trace metamodel for RDM



(a) NFRs satisficement level under normal conditions



(b) Chosen Topology

Figure 4: System behaviour - Normal conditions

i.e. the satisficement of the NFRs were in general over their thresholds. Figure 4b shows that the preferred configuration was to use a Minimum Spanning Tree Topology (MST), which helped save inter-site network costs. A query with the basic version traversal primitives showed that in 73% of the time slices, the levels of satisficement of all NFRs were over their thresholds. However, it was also detected that 24% of the time, MR was below its threshold while MP and MC were over their thresholds. Based on the observations, this behaviour could be taken as correct (see Figure 4a) as it agreed with the predefined safety tolerances, but the RDM developers wanted to know more about the reasons for this unexpected drop.

Further, the RDM developers had a theory that the proactive decision-making [37] provided by the system would sometimes make decisions which may look suboptimal at first glance (as they may reduced satisficement levels for the very first timeslices), but turned out to be optimal in the longer term (i.e. improving satisficement further the timeline). For this case study, a query based on the new primitives was used to locate those situations, to help prove the theory.

E. Query implementation

A first version of the long-term effects query was written with the full selection of primitives from Table I. The query spans 42 lines of EOL, and it has three sections:

- An operation which detects “apparently bad” decisions in the short-term (Listing 3). This operation does not use

```
operation NFRBelief shortTermNegativeAction(){
 var dec = self.eContainer;
 var blf = dec.nfrBeliefsPost
  .selectOne(blf | blf.nfr = self.nfr);
 var t = dec.rewardTable.thresholds
  .selectOne(t|t.nfr=self.nfr);
 return self.ep < t.value and self.ep > blf.ep;
}
```

Listing 3: RDM: decisions with short-term negative impact.

```
operation NFRBelief intervalInformation() {
 var intervalStart = self.unscoped.next;
 if (intervalStart.isDefined()) {
  var interval = intervalStart.sinceThen
   .before(w | w.action() <> self.action()
             or w.estimatedProbability
              < w.prev.estimatedProbability);
  if (interval.isDefined()) /* return match */
 } else {return Map {"noMatchAt"=self.time};}
}
```

Listing 4: RDM: improvement interval after decision.

```
return Decision.latest.all.first.nfrBeliefsPre
 .selectOne(blf | blf.nfr.name.contains('Rel'))
 .when(v | v.shortTermNegativeAction())
 .ifUndefined(Sequence {})
 .versions.collect(v | v.intervalInformation())
 .reject(s | s.containsKey('noMatchAt'));
```

Listing 5: RDM: main body of the query.

any temporal primitives: it only compares pre/post-action satisficement levels and the threshold.

- An operation which computes information from an interval of versions where satisficement improved from that decision (Listing 4). This version uses the primitives to define an interval of versions of the belief since the timeslice after the change (if it exists), until the timeslice before the action changed or satisficement dropped.
- The main body of the query (Listing 5). This section finds the belief for the MR NFR and the potentially bad decisions, computes the interval information for each decision, and discards decisions with no such intervals.

In order to study the impact of the temporal assertions and the scoping primitives on conciseness, the query was revised so it would only use the basic version traversal primitives. This required two changes. The first one was simple: `.when().ifUndefined().versions` in the main body was replaced with `.versions.select`. The second change was in the interval computation: the declarative use of `sinceThen` and `before` had to be replaced with a `while` loop, which also accumulated the information to be returned about the interval. In total, the EOL query went from 42 to 47 lines, which is a slight increase of around 12%, but the original intent was obscured, and the loop condition had to be refined to clip the examined history to the correct range.

The first version of the query was modified to take advantage of timeline annotation, by replacing the outermost predicate-based `when` which found the "short-term negative decisions" to use `whenAnnotated`. An initial experiment with derived attributes suggested that keeping detailed property access logs throughout the history of the model was very costly. The log-based incremental approach was replaced with a simpler approach that tested if a "bad" decision was being made for all NFRBeliefs in each version.

All three versions were tested to produce the same cases of decisions with positive longer-term consequences, in the same simulation runs of RDM. This showed that decisions with apparently immediate negative effects were producing in the long term an increase of the satisficement of the NFRs.

F. Execution times

Figure 5a includes the execution times for a first simulation run of RDM over 1000 timeslices, where after each timeslice, the temporal graph is updated in "Update" milliseconds and then the version of the query with the primitives and without timeline annotation is invoked in "Query" milliseconds. The query times include client-server communication overheads. "Total" covers the entire time the timeslice took, including simulation, temporal graph updating and querying.

Total timeslice times ranged from 1192ms to 1264ms, mostly due to the increasing query times, which took 22ms in the first query (warming up), then 3ms, and then increased up to 146ms towards the last timeslices. Graph update times remained stable throughout the timeslices, with 99% of them within 39ms and 95% of them within 28ms. While query times grew with longer histories, update times were only impacted by the size of the changes in the indexed model. RDM execution times were dominated by the simulation: on average, time-awareness took 7.5% of the time of each timeslice.

The execution times for timeline annotation are shown in Figure 5b. This approach produced similar total and update times, but shorter query times. 99% of the updates were done within 37ms and 95% of them within 20ms. Queries are faster with timeline annotation, with 99% of them within 17ms.

G. RQ1: conciseness and usability

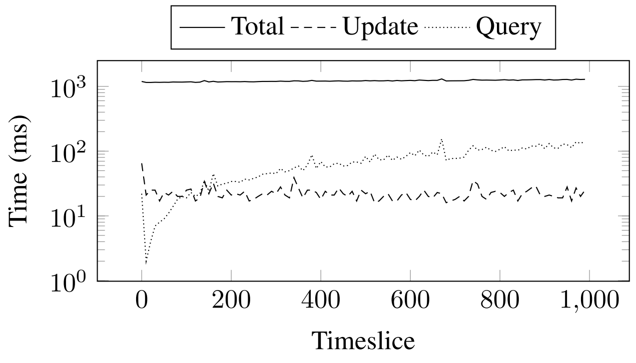
While RQ1 can only be answered on the basis of this single query, it was confirmed that dropping those primitives increased the number of lines required for the query from 42 to 47 lines. Beyond the simple numbers, it is interesting to compare the case of `when` and `sinceThen.before()`. The first primitive was easily replaced with the version traversal queries, and in fact avoided the use of `ifUndefined` to protect against the undefined value returned by `when` if no matches exist. Based on this, we are considering changing the semantics of these primitives to return empty collections when no matching versions are found, and to produce empty collections when given one. This would allow for simpler expressions with fewer special cases.

The replacement of `sinceThen.before()` required using control structures and transforming logical predicates. The predicate had to be changed so that the first version in the interval did not have its expected probability tested, as we already knew it had lower satisficement than its previous version: it came after a "bad" decision. This was done implicitly by `sinceThen`, which clipped the history to the start of the interval, but had to be reintroduced manually through timepoint comparisons in the `while`-based version. Additionally, the loop was finding the interval and computing the result at the same time, which would hinder comprehension later. This showed that the use of the primitives made writing this query less error-prone and easier to understand.

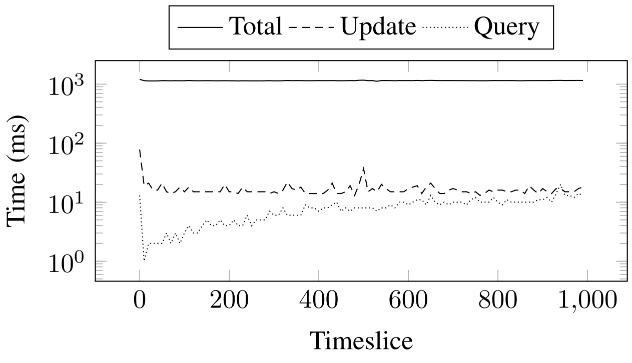
H. RQ2: performance tradeoffs

The execution times in Figure 5a show that adding time-awareness to RDM did not add more than a 12% overhead to its simulation. While update times remained stable, the cost of running the queries did increase as the history grew longer. Timeline annotation provided some noticeable improvements to the speed of the long-term effect query, which has to find a rare situation before going on: the 99th percentile went from 148.05ms without timeline annotations to 17ms with them.

There was a high cost in the incremental computation of derived features with the custom temporally consistent indices in the Greycat-based backend. They require maintaining validity intervals for each document, which is very expensive. The current timeline annotation solution relies instead on a dedicated, simpler Lucene index which does not use validity intervals. It may be worth considering how to unify these two types of pre-computed entities in the future.



(a) Without timeline annotation



(b) With timeline annotation

Figure 5: Execution times for the RDM long-term effect queries, in milliseconds using a $\log$ scale.

I. Threats to validity

There are internal and external threats to the validity of these results. In terms of internal threats, the queries were validated by the RDM domain expert through its definition and through the results obtained. However, even if the primitives have a dedicated test suite, further testing may be required to further study defects in the implementation of the primitives. The simulation was executed once per query due to time constraints: additional runs would have been useful to isolate variability from the operating system and I/O.

As for external threats, the primitives have only been tested so far on the RDM case study, and the original developers of the primitives wrote the three versions of the query. Further studies would be required on the expressiveness, usability and performance of the primitives in other case studies. One more external threat is the scalability with much longer histories. It may be necessary to limit the temporal graph to a certain time window (e.g. the last week or the last n versions) to maintain a bound on resource consumption.

V. Conclusions and future work

This paper has presented a set of extensions to a model query language for traversing the history of a model and for making temporal assertions. To provide efficient recall in future queries, the paper also presents an approach that annotates versions of interest as they are observed. The solution has been implemented in a model indexing tool, and is demonstrated using an autonomous and self-adaptive system. Further, the work presented has proved to be useful to help stakeholders to get more insights about the behaviour exposed by the running system and its decision-making.

Two research questions were raised: RQ1 considered if the notation was expressive and helped avoid mistakes, and RQ2 examined the tradeoffs imposed by adding time-awareness to a model-based approach. While the notation has only been tested on this runtime model-based case so far, the version with the primitives was 12% shorter, and not using the primitives revealed how users would be prone to common pitfalls such as forgetting to clip the query to the right intervals, or mixing interval definitions with computations of outputs. The additional processing only added about 7.5% of overhead to RDM, and even after 1000 timeslices 99% of the live queries were within 146ms. The approach was extended with the ability to mark events of interest, further dropping 99% live query times to within 17ms. This suggests that queries which need to run multiple times per second will need to limit how far they go back in time, or use timeline annotation.

There are several avenues for future work. First, the primitives will continue to be improved: the composability of the primitives can be refined to avoid adding conditional expressions when no matching versions are found, and the Occurrence patterns by Dwyer will be implemented with dedicated primitives which will be easier to read than the nested primitives currently required.

In terms of performance, the long-term scalability of the solution will be improved by providing configurations where the history is bounded in length, or by providing primitives that sample past history. Further on, it is envisioned to provide an interactive and more visual approach to studying the history of graphs. RDM is a system which uses smaller models when compared to other decision processes (e.g. Q-learning). Data-intensive decision processes may require specialised indexing and querying approaches in the future.

Providing further evidence of the general applicability of this time-aware approach is another priority. The authors intend to use these primitives and indexing solution to models developed collaboratively at design-time as well. Developers will tend to make more complex changes to their models, and the patterns to be found may be more difficult to define without the appropriate constructions.

Beyond the use of time-aware patterns for human understanding, providing further time-awareness support to autonomous systems is part of the goal of the research. If the autonomous system has the ability to study its own history, the decision-making process can be extended to consider past consequences of actions in similar situations more explicitly, to therefore improve the autonomous and reflective capabilities of the decision-making process.

Acknowledgements

The work was partially funded by the Leverhulme Trust Research Fellowship RF-2019-548 of Dr Bencomo.

References

[1] A. Gómez, J. Cabot, and M. Wimmer, "TemporalEMF: A temporal metamodeling framework," in *Conceptual Modeling*, J. C. Trujillo, K. C. Davis, X. Du, Z. Li, T. W. Ling, G. Li, and M. L. Lee, Eds. Cham: Springer International Publishing, 2018, pp. 365–381.

[2] A. Benelallam, T. Hartmann, L. Mouline, F. Fouquet, J. Bourcier, O. Barais, and Y. Le Traon, "Raising time awareness in model-driven engineering: Vision paper," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2017, pp. 181–188.

[3] A. García-Domínguez, N. Bencomo, and L. H. G. Paucar, "Reflecting on the past and the present with temporal graph-based models," in *Proceedings of MODELS 2018 Workshops*, vol. 2245. Denmark: CEUR-WS.org, 2018, pp. 46–55.

[4] A. Artale, C. Parent, and S. Spaccapietra, "Evolving objects in temporal information systems," *Annals of Mathematics and Artificial Intelligence*, vol. 50, no. 1, pp. 5–38, Jun 2007. [Online]. Available: https://doi.org/10.1007/s10472-007-9068-z

[5] A. Artale, R. Kontchakov, V. Ryzhikov, and M. Zakharyaschev, "A cookbook for temporal conceptual data modelling with description logics," *ACM Trans. Comput. Logic*, vol. 15, no. 3, pp. 25:1–25:50, Jul. 2014.

[6] N. Bencomo, S. Götz, and H. Song, "Models@run.time: a guided tour of the state of the art and research challenges," *Software & Systems Modeling*, Jan 2019. [Online]. Available: https://doi.org/10.1007/s10270-018-00712-x

[7] L. Mouline, A. Benelallam, F. Fouquet, J. Bourcier, and O. Barais, "A temporal model for interactive diagnosis of adaptive systems," in *2018 IEEE International Conference on Autonomic Computing, ICAC 2018, Trento, Italy, September 3-7, 2018*, 2018, pp. 175–180.

[8] N. Bencomo, "Quantun: Quantification of uncertainty for the reassessment of requirements," *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pp. 236–240, 2015.

[9] N. Bencomo and A. Belaggoun, "A world full of surprises: Bayesian theory of surprise to quantify degrees of uncertainty," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 460–463. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591118

[10] Object Management Group, "Object Constraint Language Specification 2.4," Specification, Feb. 2014, date of last access: 29th April 2019. Archived in http://archive.is/gTkkG. [Online]. Available: https://www.omg.org/spec/OCL/2.4/

[11] D. S. Kolovos, R. F. Paige, and F. Polack, "The Epsilon Object Language (EOL)," in *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, 2006, pp. 128–142.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of ICSE'99*, May 1999, pp. 411–420.

[13] Object Management Group, "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) 1.2 Beta1," Dec. 2018, archived at http://archive.is/UaK85. [Online]. Available: https://www.omg.org/spec/MARTE/1.2/Beta1/

[14] P. Ziemann and M. Gogolla, "An extension of OCL with temporal logic," in *Critical Systems Development with UML*, vol. 2, 2002, pp. 53–62.

[15] B. Kanso and S. Taha, "Specification of temporal properties with OCL," *Science of Computer Programming*, vol. 96, pp. 527–551, Dec. 2014.

[16] F. Hilken and M. Gogolla, "Verifying Linear Temporal Logic Properties in UML/OCL Class Diagrams Using Filmstripping," in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug. 2016, pp. 708–713.

[17] W. Dou, D. Bianculli, and L. Briand, "A Model-Driven Approach to Trace Checking of Pattern-Based Temporal Properties," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2017, pp. 323–333.

[18] A. Benelallam, T. Hartmann, L. Mouline, F. Fouquet, J. Bourcier, O. Barais, and Y. L. Traon, "Raising Time Awareness in Model-Driven Engineering: Vision Paper," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2017, pp. 181–188.

[19] B. Meyers, H. Vangheluwe, J. Denil, and R. Salay, "A Framework for Temporal Verification Support in Domain-Specific Modelling," *IEEE Transactions on Software Engineering*, 2018.

[20] Software Freedom Conservancy, "Git," Feb. 2019, date of last access: April 28th, 2019. Archived in http://archive.is/DD6qG. [Online]. Available: https://git-scm.com/

[21] Eclipse Foundation, "CDO Model Repository," date of last access: April 28th, 2019. Archived at http://archive.is/nYpNb. [Online]. Available: http://www.eclipse.org/cdo/

[22] C. Debreceni, G. Bergmann, M. Búr, I. Ráth, and D. Varró, "The MONDO collaboration framework: secure collaborative modeling over existing version control systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. Paderborn, Germany: ACM Press, 2017, pp. 984–988.

[23] K. Barmpis and D. S. Kolovos, "Towards Scalable Querying of Large-Scale Models," in *Proceedings of ECMFA'14*, 2014, pp. 35–50.

[24] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*, 2nd ed. O'Reilly, 2015, ISBN 978-1-4919-3089-2.

[25] A. Hegedüs, G. Bergmann, C. Debreceni, A. Horváth, P. Lunk, A. Menyhért, I. Papp, D. Varró, T. Vileiniskis, and I. Ráth, "Incquery Server for TeamWork Cloud: Scalable Query Evaluation over Collaborative Model Repositories," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '18. New York, NY, USA: ACM, 2018, pp. 27–31, event-place: Copenhagen, Denmark. [Online]. Available: http://doi.acm.org/10.1145/3270112.3270125

[26] I. Dávid, I. Ráth, and D. Varró, "Foundations for Streaming Model Transformations by Complex Event Processing," *Software & Systems Modeling*, vol. 17, no. 1, pp. 135–162, Feb. 2018.

[27] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot, "NeoEMF: A multi-database model persistence framework for very large models," *Science of Computer Programming*, vol. 149, pp. 9–14, Dec. 2017.

[28] V. Z. Moffitt and J. Stoyanovich, "Temporal graph algebra," in *Proceedings of the 16th International Symposium on Database Programming Languages - DBPL '17*. Munich, Germany: ACM Press, 2017, pp. 1–12.

[29] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. L. Traon, "Analyzing Complex Data in Motion at Scale with Temporal Graphs," in *Proceedings of SEKE'17*, Jul. 2017, pp. 596–601.

[30] M. Haeusler, T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu, "ChronoGraph: A Versioned TinkerPop Graph Database," in *Data Management Technologies and Applications*, ser. Communications in Computer and Information Science, J. Filipe, J. Bernardino, and C. Quix, Eds. Springer International Publishing, 2018, pp. 237–260.

[31] Lars George, *HBase: The Definitive Guide*, 2nd ed. O'Reilly Media, Aug. 2017, ISBN 978-1-4920-2425-5.

[32] K. Barmpis, S. Shah, and D. S. Kolovos, "Towards Incremental Updates in Large-Scale Model Indexes," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, G. Taentzer and F. Bordeleau, Eds. Springer International Publishing, Jul. 2015, no. 9153, pp. 137–153.

[33] G. Blair, N. Bencomo, and R. France, "Guest editor's introduction: Models@run.time," *IEEE Software*, 2009.

[34] M. Ji, A. C. Veitch, J. Wilkes *et al.*, "Seneca: remote mirroring done write." in *USENIX Annual Conference*, 2003, pp. 253–268.

[35] A. Ramirez, B. Cheng, N. Bencomo, and P. Sawyer, "Relaxing claims: Coping with uncertainty while evaluating assumptions at run time," *MODELS*, 2012.

[36] E. M. Fredericks, "Mitigating uncertainty at design time and run time to addressassurance for dynamically adaptive systems," *Michigan State University. PhD Thesis.*, 2015.

[37] N. Ye, A. Somani, D. Hsu, and W. S. Lee, "DESPOT: online POMDP planning with regularization," *J. of Artificial Intelligence Research*, 2017.