TemporalEMF: A Temporal Metamodeling Framework*

Abel Gómez[1][0000−0003−1344−8472], Jordi Cabot[1,2][0000−0003−2418−2489], and Manuel Wimmer[3][0000−0002−1124−7098]

[1] Internet Interdisciplinary Institute (IN3), Universitat Oberta de Catalunya (UOC), Spain
agomezlla@uoc.edu
[2] ICREA, Spain
jordi.cabot@icrea.cat
[3] CDL-MINT, TU Wien, Austria
wimmer@big.tuwien.ac.at

Abstract. Existing modeling tools provide direct access to the most current version of a model but very limited support to inspect the model state in the past. This typically requires looking for a model version (usually stored in some kind of external versioning system like Git) roughly corresponding to the desired period and using it to manually retrieve the required data. This approximate answer is not enough in scenarios that require a more precise and immediate response to temporal queries like complex collaborative co-engineering processes or runtime models. In this paper, we reuse well-known concepts from temporal languages to propose a temporal metamodeling framework, called *TemporalEMF*, that adds native temporal support for models. In our framework, models are automatically treated as temporal models and can be subjected to temporal queries to retrieve the model contents at different points in time. We have built our framework on top of the Eclipse Modeling Framework (EMF). Behind the scenes, the history of a model is transparently stored in a NoSQL database. We evaluate the resulting *TemporalEMF* framework with an Industry 4.0 case study about a production system simulator. The results show good scalability for storing and accessing temporal models without requiring changes to the syntax and semantics of the simulator.

Keywords: Temporal Models, Metamodeling, Model-Driven Engineering

1 Introduction

Modeling tools and frameworks have improved drastically during the last decade due to the maturation of metamodeling concepts and techniques [9]. A concern which did not yet receive enough attention is the temporal aspect of metamodels and their corresponding models when it comes to model valid time and transaction time dimensions instead of just arbitrary user-defined times [15]. Indeed, existing modeling tools provide direct

* This work was supported by the *Austrian Federal Ministry for Digital, Business and Enterprise* and by the *National Foundation for Research, Technology and Development*; the *Programa Estatal de I+D+i Orientada a los Retos de la Sociedad* Spanish program (Ref. TIN2016-75944-R); and the *Electronic Component Systems for European Leadership Joint Undertaking* under grant agreement No. 737494. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation program and from Sweden, France, Spain, Italy, Finland & Czech Republic.

access to the most current version of a model, but very limited support to inspect the model state at specific past time periods [5,8]. This typically requires looking for a model version stored in some kind of model repository roughly corresponding to that time period and using it to manually retrieve the required data. This approximate answer is not enough in scenarios that require a more precise and immediate response to temporal queries like complex collaborative co-engineering processes or runtime models [20].

To deal with these new scenarios, temporal language support must be introduced as well as an infrastructure to efficiently manage the representation of both historical and current model information. Furthermore, query means are required to validate the evolution of a model, to find interesting modeling states, as well as execution states. Using existing technology to tackle these requirements is not satisfactory as we later discuss.

To tackle these limitations, we reuse well-known concepts from temporal languages to propose a temporal metamodeling framework, called *TemporalEMF*, that adds native temporal support. In *TemporalEMF*, models are automatically treated as temporal, and temporal query support allows to retrieve model elements at any point in time. Our framework is realized on top of the Eclipse Modeling Framework (EMF) [24]. Models history is transparently stored in a NoSQL database, thus supporting large evolving models. We evaluate the resulting framework using an Industry 4.0 case study of a production system simulator [19]. The results show good scalability for storing and accessing temporal models without requiring changes to the syntax and semantics of the simulator.

Thus, our contribution is three-fold: (*i*) we present a **light-weight extension** of current metamodeling standards to build a temporal metamodeling language; (*ii*) we introduce an **infrastructure to manage temporal models** by combining EMF and HBase [26], an implementation of Google's BigTable NoSQL storage [12]; and (*iii*) we outline a **temporal query language** to retrieve historical information from models. Please note that contributions do not change the general way how models are used: if only the latest state is of interest, the model is transparently accessed and manipulated in the standard way as offered by the EMF. Thus, all existing tools are still applicable, and the temporal extension is considered to be an add-on.

This paper is structured as follows. Section 2 presents our proposal to include temporal information in existing metamodeling standards. Section 3 presents how temporal models can be stored in a NoSQL database, and Section 4 presents our prototype based on EMF and HBase. Our approach is evaluated with a case study in Section 5. Section 6 presents related work, and Section 7 concludes the paper with an outlook on future work.

2 Temporal Metamodeling

In this section, we discuss how existing work on temporal modeling can be applied for temporal metamodeling. We introduce a profile for adding temporal concepts in existing metamodeling standards; and we present an Industry 4.0 case which demands for temporal metamodeling in order to realize simulation and runtime requirements.

2.1 A Profile for Temporal Metamodeling

We propose a profile for augmenting existing metamodels with information about temporal aspects. Metamodels can be regarded as just a special kind of models [7], and

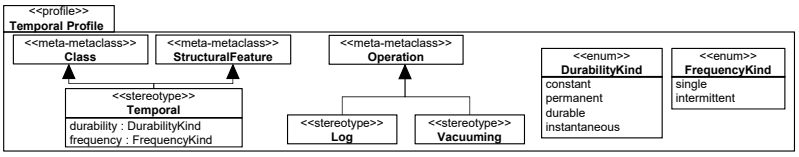

Fig. 1: Profile for Temporal Metamodeling.

therefore, existing work on temporal modeling for ER [15] and UML [11] languages can be easily leveraged to specify arbitrary temporal (meta)models. Thus, we base our temporal metamodeling profile on these previous works for the static parts of the model, and extend them to cover behavioural definitions which are of particular interested if executable metamodels, i.e., executable modeling languages, are used.

Figure 1 introduces the profile for augmenting metamodels with temporal concerns in EMF Profiles notation. EMF Profiles [18] is a generalization of UML Profiles. As with UML Profiles, stereotypes are defined for predefined metaclasses and represent a way to provide lightweight extensions for modeling languages without requiring any changes to the technological infrastructure. However, in contrast to UML Profiles, EMF Profiles allow modelers to define profiles for any kind of modeling language.

The profile in Figure 1 includes the stereotype *Temporal* (inspired from previous work on temporal UML [11]) combined with its *durability* and *frequency* properties to classify metaclasses as *instantaneous*, *durable*, *permanent* or *constant*. We also introduce novel stereotypes for annotating the operations as we consider executable metamodels. We want to define special operations for which their calls are *logged* or *vacuumed*. The former requires to keep a trace of all executions of the operation. The later forces to restart from scratch the lifespan of the modeling elements deleting their complete previous history. This should be obviously used with caution as it defeats the purpose of having the temporal annotations in the first place but it may be necessary in scenarios where runtime models are used for simulation purposes and modelers want to restart that simulation using a clean slate. Furthermore, it may help in managing the size of temporal models which may attach an extensive history where only the last periods are of interest.

Similarly, we have also adapted our previous work on the specification of temporal expressions [11] to provide temporal OCL-based query support on top of our temporal infrastructure. Before we show an application of the profile and query support, we introduce the motivating and running example of this paper.

2.2 Running Example: Transportation Line Modeling Language

The running example of the paper is taken from the CDL-MINT[a] project. The main goal is to investigate the application of modeling techniques in the domain of smart production systems. The example is about designing transportation lines made up of sets of turntables, conveyors, and multi-purpose machines. The production plant is supposed to continuously processes items by its multi-purpose machines located in specific areas. Turntables and conveyors are in charge of moving items to these machines.

[a] More information available at: `https://cdl-mint.big.tuwien.ac.at`

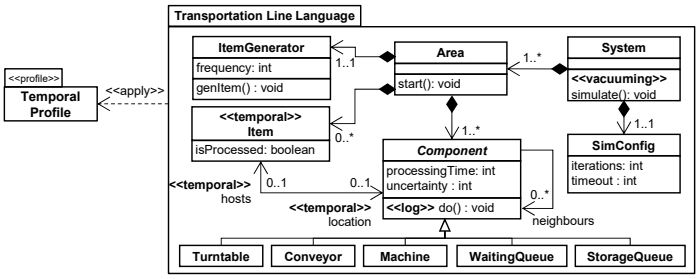

Fig. 2: Applying the Temporal Profile for a Transportation Line Modeling Language

Given a particular design for a transportation line, simulations are needed for computing different KPIs such as utilization, throughput, cycle times, etc., in order to validate if certain requirements are actually met by a particular design.

The metamodel of the transportation line modeling language is shown in Fig. 2. *System* is the root class, which is composed of several *Area*s. The system has associated a *SimConfig*, where parameters of the simulation can be specified, e.g., the simulation time or number of iterations. An area, in turn, can contain any number of *Component*s. As we see, there are five types of *Component*s, namely, *Conveyor*, *Machine*, *Turntable*, *StorageQueue*, and *WaitingQueue*. Items are created by *ItemGenerator*s and are moved along the transportation line by starting their way in the area's associated *WaitingQueue*. On their way, they may serve as input to *Machine*s. Those items that complete the transportation line successfully, should end up in a *StorageQueue*.

In Fig. 2, we also provide an application example of the introduced temporal profile. In particular, we mark the class *Item* as *Temporal* as instances of this class are created during runtime which should be tracked in the history of the model. Furthermore, not only the items, but also their assignment to particular locations should be tracked. For this, we also annotate the bi-directional reference between *Item* and *Component*. In order to understand which component is activated in a particular point in time, we annotate the *do()* operation with the *Log* stereotype. Finally, in order to create a fresh state when a simulation run is started, we annotate the *simulation()* operation with the *Vacuuming* stereotype.

Having the class *Item* marked as a temporal element as well as the involved references, we are now able to define several queries (**Qs**) to compute execution states of interest (such as those needed for provenance) and KPIs (such as utilization):

Q1 — Find all items which have been processed by machine *m*.
Q2 — Find the components which had an item assigned at a particular point in time.
Q3 — Find the components which had an item assigned within a particular time frame.
Q4 — Compute the utilization of machine *e* for the whole system execution lifecycle.

Query **Q1** retrieves the complete evolution of a structural feature, namely the hosts reference. **Q2** accesses the hosts reference for a particular point in time, while **Q3** is evaluating this reference for any particular moment between two time instants. Finally, **Q4** is performing a complex query which is also requiring the access of the time values for having items assigned and not having items assigned.

As an example of how these queries are defined using a temporal OCL [21] extension, below we find the specification for **Q2**. In particular, by using additional access methods for properties which are time sensitive (cf. *hostsAt(i:Instant)*) we are able to query the state of the hosts reference for a particular moment in the past.

```
1 Component.allInstances()->select(c | not c.hostsAt(instant).oclIsUndefined())
```

3 Approach

Enabling a temporal metamodeling language as the one discussed above requires a temporal modeling infrastructure. In this section, we introduce the core concepts of our solution, based on the use of a key-value NoSQL mechanism to store the models' historical data. Next section gives additional technical details on its design.

In a previous work [14], we discussed why NoSQL data stores and, more concretely, map-based (i.e. key-value) stores are especially well-suited to persist models managed by (meta-)modeling frameworks since map-based stores are very well aligned with the typical fine-grained APIs offered by modeling frameworks (that mostly force individual access to model elements, even when the user aims to query a large subset of the model). Alternative mechanisms, such as in-memory or XML-based, failed to scale when dealing with large models as it typically happens when working on, for instance, building information models (BIMs), modernization projects involving the model-based reengineering of legacy systems, or on simulation scenarios. This is also true for relational databases (even temporal ones, a direction they are all following in compliance with the SQL:2011 standard) mainly due to the lack of alignment with modeling tools APIs.

An interesting map-based solution is BigTable [12]. BigTable is a distributed, scalable, versioned, non-relational and column-based big data store; where data is stored in tables, which are sparse, distributed, persistent, and multi-dimensional sorted maps. These maps are indexed by the tuple **row key, column key, and a timestamp**. The native presence of timestamps and the benefits of map-based solutions to store large models make a BigTable-like solution an ideal candidate for a temporal modeling infrastructure.

Next, we describe BigTable's main concepts and how we adapt them (and, in general, similar column-based solutions) to support a temporal modeling infrastructure able to automatically persist and manage (meta)models annotated with our profile.

3.1 BigTable Basics

The top-level organization units for data in BigTable are named *tables*; and within tables, data is stored in *rows*, which are identified by their *row key*. Within a row, data is grouped by *column families*, which are defined at table creation. All rows in a table have the same column families, although may be empty. Data within a column family is addressed via its *column qualifier*; which, on the contrary, do not need to be specified in advance nor be consistent between rows. Cells are identified by their *row key*, *column family*, and *column qualifier*, do not have a data type, and store raw data which is always treated as a `byte[]`. Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by BigTable is configurable for each column family.

3.2 Column-based Data Model

Our proposed data model flattens the typical graph structure expressed by models into a set of key-value mappings that fit the map-based data model of BigTable. Such data model takes advantage of unique identifiers that are assigned to each model object.

Fig. 3a shows a simplification of the production plant model for the case study presented in Section 2.2 that we will use as an example. The figure describes a production system (omitted for the sake of simplicity) with a single *Area* with *machines*, which in turn, *host* and *process* – one by one – a set of *items* that are fed into the production system. Figs. 3b-3d, present three instances of this model in three different consecutive instants. Changes performed at each instant are highlighted in red. Fig. 3b represents an area *a1*, at a given moment in time t_i, with one machine *m1*, and one unprocessed item *i1*. Fig. 3c represents the same area at time t_{i+1}, when item *i1* – which is ready to be processed – is fed into *m1*. Finally, Fig. 3d represents the area at time t_{i+2}, once *m1* has processed *i1*, thus changing the *isProcessed* status to `true`.

Our proposed data model uses a single table with three column families to store models' information: (i) a *property column family*, that keeps all objects' data stored together; (ii) a *type column family*, that tracks how objects interact with the meta-level (such as the *instance of* relationships); and (iii) a *containment column family*, that defines the models' structure in terms of containment references. Table 1 shows how the sample instances in Figs. 3b-3d are represented using this structure.

As Table 1 shows, row keys are the objects' *unique identifiers*. The PROPERTY *column family* stores the objects' actual data. Please note that not all rows have a value for a given column (as BigTable tables are *sparse*). How data is stored depends on the *property type* and *cardinality* (i.e., upper bound). For example, values for single-valued attributes (like the *id*, which is stored in the ID column) are directly saved as a single literal value; while values for many-valued attributes are saved as an array of single literal values (Fig. 3 does not contain an example of this). Values for single-valued references, such as the *hosts* reference from *Machine* to *Item*, are stored as a single value (corresponding to the identifier of the referenced object). Finally, multi-valued references are stored as an array containing the literal identifiers of the referenced objects. Examples of this are the *machines* and *items* containment references, from *Area* to *Machine* and *Item*, respectively.

As it can be seen, the table keeps track of all current and past model states. At t_i (cf. Fig. 3b), the model is stored in rows $\langle$`'ROOT'`$, t_i\rangle$, $\langle$`'a1'`$, t_i\rangle$, $\langle$`'m1'`$, t_i\rangle$ and $\langle$`'i1'`$, t_i\rangle$.

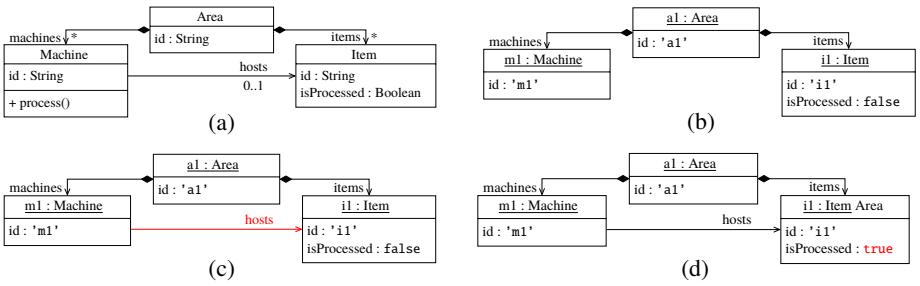

Fig. 3: Example model (3a) and sample instances at t_i (3b), t_{i+1} (3b) and t_{i+2} (3b)

Table 1: Example model stored in a sparse table in BigTable

		PROPERTY					
KEY	TIMESTAMP	ROOTCONTENTS	ID	MACHINES	ITEMS	HOSTS	ISPROCESSED
'ROOT'	t_i	{ 'a1' }	—	—	—	—	—
'a1'	t_i	—	'a1'	{ 'm1' }	{ 'i1' }	—	—
'm1'	t_{i+1}	—	'm1'	—	—	'i1'	—
'm1'	t_i	—	'm1'	—	—	—	—
'i1'	t_{i+2}	—	'i1'	—	—	—	true
'i1'	t_i	—	'i1'	—	—	—	false

(continued)		CONTAINMENT		TYPE	
KEY	TIMESTAMP	CONTAINER	FEATURE	NSURI	ECLASS
'ROOT'	t_i	—	—	'http://plant'	'RootEObject'
'a1'	t_i	'ROOT'	'rootContents'	'http://plant'	'Area'
'm1'	t_{i+1}	'a1'	'machines'	'http://plant'	'Machine'
'm1'	t_i	'a1'	'machines'	'http://plant'	'Machine'
'i1'	t_{i+2}	'a1'	'items'	'http://plant'	'Item'
'i1'	t_i	'a1'	'items'	'http://plant'	'Item'

After setting the *hosts* reference at instant t_{i+1} (cf. Fig. 3c), the new $\langle$'m1', $t_{i+1}\rangle$ row – which supersedes $\langle$'m1', $t_i\rangle$ – is added. When the *isProcessed* property is changed (cf. Fig. 3d), the $\langle$'i1', $t_{i+2}\rangle$ row is added; and the last model state is stored in rows $\langle$'ROOT', $t_i\rangle$, $\langle$'a1', $t_i\rangle$, $\langle$'m1', $t_{i+1}\rangle$ and $\langle$'i1', $t_{i+2}\rangle$. Note that our infrastructure is not bitemporal: we assume that valid-time and transaction-time are always equivalent.

Structurally, EMF models are trees, and thus, every non-volatile *object* (except the root *object*) must be contained within another *object* (i.e., referenced via a containment *reference*). The CONTAINMENT *column family* maintains this information for every persisted object at a specific instant in time. The CONTAINER column stores the identifier of the container object, while the FEATURE column indicates the *property* that relates the container object with the child object. Table 1 shows that, for example, the container of the *Area a1* is ROOT through the *rootContents* property (i.e., it is a root object and is not contained by any other object). In the next row we find the entry that describes that the *Machine m1* is contained in the *Area a1* through the *machines property*.

The TYPE *column family* groups the type information by means of the NSURI and ECLASS columns. For example, the table specifies the element *a1* is an instance of the *Area* class of the *Plant* metamodel (that is identified by the http://plant NSURI). Data stored in the TYPE *column family* is immutable and never changes.

3.3 Query facilities

As mentioned in Section 2, several temporal query languages have been proposed before. Nevertheless, they all share the need to refer to the value of an attribute or an association at a certain (past) instant of time *i* in order to evaluate the temporal expressions [11] (also known as temporal interpolation functions). Based on this general requirement, we have built the generic *TObject::eGetAt(i:instant, f:feature)* method that returns the value of a feature (either an attribute or an association end) for a specific *temporal object* at a specific instant. For convenience, we also provide *TObject::eGetAllBetween(s:startInstant, e:endInstant, f:feature)*, that returns a sorted map where the key of the map is the moment when the feature was updated, and the value is the value that was set at that specific moment within the given period. In Section 2.2, we showed how **Q2** could be expressed

in temporal OCL. As an example, below we depict how to specify such query for a specific *Area a1* in our proposed Java-based query language. This language makes use of the EMF Java API [24], taking advantage of Java streams and lambda expressions.

```
a1.getComponent().stream()
.filter(c -> c.eGetAt(instant, TllPackage.eINSTANCE.getComponent_Hosts()) != null)
.collect(Collectors.toSet());
```

4 *TemporalEMF* Architecture

We have built our temporal (meta-)modeling framework on top of Apache HBase [26], the most wide-spread open-source implementation of BigTable, based on our experience on building scalable, non-temporal model persistence solutions [6].

Fig. 4 shows the high-level architecture of our proposal. It consists of a *temporal model management interface* – *TemporalEMF* – built on top of a regular *model management interface* – EMF [24]. These interfaces use a *persistence manager* in such a way that tools built over the temporal (meta)modeling framework would be unaware of it. The persistence manager communicates with the underlying database by a *driver*. In particular we implement *TemporalEMF* as a persistence manager on top of HBase; but other persistence technologies can be used as long as a proper driver is provided.

Thanks to our identifier-based data model, *TemporalEMF* offers lightweight on-demand loading and efficient garbage collection. Model changes are automatically reflected in the underlying storage. To do so, (i) we decouple dependencies among objects taking advantage of the *unique identifier* assigned to all model objects. (ii) We implement an *on-demand loading and saving* mechanism for each live model object by creating a thin delegate object that is in charge of on-demand loading the element data from storage and keeping track of the element's state. Data is loaded/saved from/to the persistence backend by using the object's unique identifier. Finally, and thanks to the data model explained in Section 3.2. (iii) We provide a *garbage collection-friendly* implementation where no hard references among model objects are kept, so that any model object that is not directly referenced by the application can be deallocated.

TemporalEMF is designed as a simple persistence layer that adds temporal support to EMF. As in standard EMF, no thread-safety is guaranteed, and no transactional support is explicitly provided, although all ACID properties [16] are guaranteed at the object level. Nevertheless, we paid special attention to keep the same semantics than in basic EMF. Thus, *TemporalEMF*, available as an open-source project [1], can be directly plugged into any EMF-based tool to immediately provide enhanced temporal support.

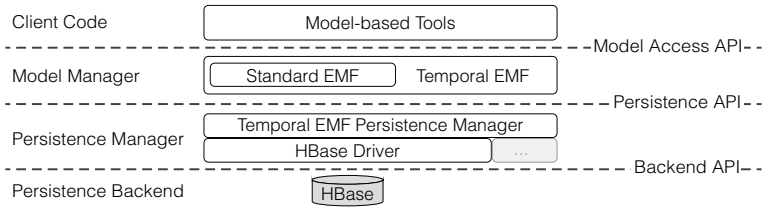

Fig. 4: Overview of the model-persistence framework

5 Evaluation

In this section, we perform experiments based on the guidelines for conducting empirical explanatory case studies [22]. The main goal is to evaluate the impact of the temporal extension for models on the performance as well as the capabilities of temporal queries in the context of model-based simulations. All the artifacts used in this evaluation and all the data we have gathered (either raw or processed) can be inspected at the paper web page[b].

This study aims to evaluate the possible distinct behavior of current in-memory solutions without dedicated temporal support (in the following, *StandardEMF*) and our temporal solution (in the following, *TemporalEMF*) and, more specifically, to answer the following research questions (**RQs**):

RQ1: Production Cost — Is there a significant difference of the time required for producing and manipulating temporal model elements? This question is of particular relevance for efficient model simulators which have to run for longer periods to produce a target model as well as the traces to reach such a model.

RQ2: Storage Cost — Is there a significant difference of the storage size of temporal models? This question is of particular interest since the output of a model simulation may have to be stored for provenance reasons or for comparing different variants.

RQ3: Reproduction Cost — Is there a significant difference of restoring previous versions of temporal model elements? This question is of relevance as the different properties of a simulation run have to be computed which involves accessing past states and information of the simulated model.

5.1 Case Study Setup

Next, we summarize the selected case study, the input models, the evaluation measures and the environment used to perform the evaluation.

Selected Case: Transportation Line Simulator — Section 2.2, where we exemplify the application of the temporal modeling profile, already presents the case we use in this evaluation: the Transportation Line Modeling Language. However, in order to have a reference implementation with which to compare *TemporalEMF*, we need to extend the metamodel so that *StandardEMF* can also provide temporal capabilities. Following a naive approach, we introduce additional metamodel elements (as well as listeners), so that the history of our model elements can be explicitly tracked and stored in the model itself.

Fig. 5 shows an example of such additions: the *ItemHistoryEntry* metaclass can be added to the metamodel so that everytime the *hosts* property of a *Component* changes, the

[b] `http://hdl.handle.net/20.500.12004/1/C/ER/2018/043`

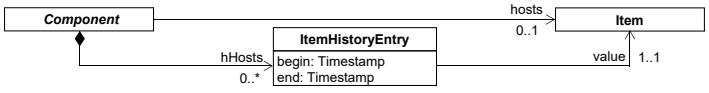

Fig. 5: Extension of the Transportation Line Modeling Language for *StandardEMF*

corresponding *ItemHistoryEntry* element is created and/or updated. Following this naive approach, we add as many extensions to the original metamodel as many temporal properties we aim to track (in our case, the *Component::hosts* and *Item::location* properties).

Input Models — In all the experiments we have used a model with a single *System* and (as we well see later in Section 5.2) varying *SimConfig* parameters depending on the RQ to be answered. For reference, this *System* is composed by a single *Area*, with 1 *ItemGenerator*, 1 *WaitingQueue*, 1 *StorageQueue*, 3 *Conveyors*, 4 *TurnTables*, and 1 *Machine*.

For **RQ1**, we execute the simulation varying the processing times for the elements of the *Area* from 0 ms (no processing time), to 40 ms, following an arithmetic progression with a common difference of 5 milliseconds. Additionally, we run the experiments in simulations whose duration (measured in the number of iterations executed) varies from 20 iterations to 40 960 iterations, following a geometric progression with a common ratio of 2. To answer **RQ2**, we only vary the duration of the simulation and the amount of memory available in the system; as well as we do for **RQ3**.

Evaluation Measures — We use different metrics depending on the nature of the research question. For **RQ1**, we measure the execution time needed for running the same simulation both using *StandardEMF* and *TemporalEMF*. We vary both the total simulation time (iterations), and the processing time in the transportation line model to evaluated how the different simulation parameters impact on the execution time. The procesing time can be considered the *think time* that determines the workload we apply to the simulation execution. To evaluate **RQ2**, we measure both the used memory during the simulation as well as the storage needed to keep the simulation outcomes for both solutions. To evaluate **RQ3**, we measure the execution time for recreating previous versions of model elements for both solutions. Specifically, we execute the code implementing queries **Q1**–**Q4** presented in Section 2.2.

Environment Setup — We have executed the experiments using two Linux containers in a Proxmox VE 5.1-46 server: one for running the EMF-based code, and other for running HBase. Each one had 8 GB of RAM and 4 virtual CPUs. The actual hardware is a Fujitsu Primergy RX200 S8 server with two quad-core Intel Xeon E5-2609 v2 CPUs at 2.50GHz, 48 GB of DDR3 RAM memory (1 333 MHz), and two hard disks (at 7 200 rpm) configured in a software-controlled RAID 1. The experiments were run using Java OpenJDK 1.8.0_162, Ant 1.9.9, Eclipse Oxygen 4.7.3a, EMF 2.13 and HBase 1.4.0.

5.2 Result Analysis

Below we summarize our experiments. For the sake of brevity, only the most significant results are shown. For a comprehensive report please refer to the paper web page.

RQ1 — Table 2 shows the results for the experiments executed to evaluate the *Production Cost*. *TemporalEMF* imposes an overhead that is especially noticeable when models are small and there is no time between model modification (i.e., processing time is zero). In those extreme cases, *TemporalEMF* is up to $\sim$23 times slower (i.e., it has an overhead of 2291%). This is understandable, since such small models are completely loaded in memory in *StandardEMF*. On the contrary, on *TemporalEMF*, every single model operation implies a database access thus imposing a big cost.

Table 2: Execution times (in seconds) for the experiments for RQ1

PROC.	5120 ITERATIONS			10240 ITERATIONS			2048 ITERATIONS			40960 ITERATIONS		
TIME	SEMF	TEMF	%	SEMF	TEMF	%	SEMF	TEMF	%	SEMF	TEMF	%
0 ms	11	261	2 291%	39	588	1 405%	99	1 167	1 078%	367	2 488	578%
5 ms	102	377	268%	217	738	240%	492	1 491	203%	1 169	3 627	210%
10 ms	193	473	145%	392	939	140%	839	1 917	128%	1 902	4 328	128%
15 ms	282	620	120%	570	1 219	114%	1 206	2 227	85%	2 593	4 752	83%
20 ms	373	711	91%	755	1 301	72%	1 556	2 592	67%	3 314	6 236	88%
25 ms	461	801	74%	941	1 606	71%	1 939	3 213	66%	4 004	6 616	65%
30 ms	552	841	52%	1 113	1 683	51%	2 275	3 443	51%	4 738	7 492	58%
35 ms	641	942	47%	1 297	1 880	45%	2 640	3 761	42%	5 472	8 279	51%
40 ms	731	1 033	41%	1 471	2 084	42%	3 012	4 158	38%	6 162	9 149	48%

Table 3: Memory usage and disk usage (in MB) for the experiments for RQ2

	MEMORY USAGE (MAX HEAP 512 MB)		MEMORY USAGE (MAX HEAP 2 GB)		DISK USAGE	
ITS.	SEMF	TEMF	SEMF	TEMF	SEMF	TEMF
5 120	19	9	20	9	6	11
10 240	32	9	34	10	12	21
20 480	61	10	64	12	23	41
40 960	112	11	118	13	45	82
81 920	219	15	225	16	89	164
163 840	(i)	22	438	22	177	328
327 680	(i)	28	865	36	354	656
655 360	(ii)	(ii)	(i)	37	(i)	1 341

(i) Out Of Memory Error; (ii) Setup not executed because experiment was already failing in *StandardEMF* for smaller sizes.

Table 4: Execution times (in milliseconds) for the queries for RQ3

	QUERY EXECUTION TIME (MAX HEAP 512 MB)								QUERY EXECUTION TIME (MAX HEAP 2 GB)							
	SEMF				TEMF				SEMF				TEMF			
ITS.	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
5 120	22	13	8	6	145	11	17	65	13	12	8	6	107	10	16	66
10 240	23	24	20	10	225	14	29	93	23	23	15	10	157	11	19	79
20 480	37	39	29	18	1 694	17	21	138	38	36	31	18	239	10	22	172
40 960	43	49	40	35	5 492	31	32	318	47	44	35	34	1 438	23	24	257
81 920	79	63	60	45	14 453	47	35	602	69	56	50	41	11 570	28	21	417
163 840	(i)	(i)	(i)	(i)	34 035	42	23	1 510	93	79	74	52	28 913	36	30	852
327 680	(i)	(i)	(i)	(i)	68 839	34	20	2 262	184	128	122	74	61 101	33	28	2 163
655 360	(ii)	(ii)	(ii)	(ii)	(ii)	(ii)	(ii)	(ii)	(i)	(i)	(i)	(i)	135 975	36	36	3 595

(i) Out Of Memory Error; (ii) Setup not executed because experiment was already failing in *StandardEMF* for smaller sizes.

As the simulation time increases and the model size grows, the overhead is drastically reduced: by just increasing the procesing time from 0 to 5 ms, the overhead is reduced by 10 times (i.e. only $\sim$2 times slower). When the processing time is higher than 30 ms, the overhead is reduced to only $\sim$0.5 times slower. These numbers remain stable with increasing simulation times. It is worth noting that the overhead is only noticeable when modifications happen in the range of ms. Thus, in activities where modifications happen in the range of seconds (e.g. collaborative modeling) the overhead is unnoticeable.

RQ2 — Table 3 shows the evaluation for *Storage Cost*. The table summarizes how much memory and storage space is used after running a simulation. To measure the memory consumption, the whole simulation process is executed, the resulting models are saved in disk (in XMI for *StandardEMF*, in HBase for *TemporalEMF*), and after requesting the garbage collector for three times, the actual used memory is measured. As expected, *StandardEMF* uses much more RAM than *TemporalEMF* since all model states are kept in memory; and as it can be observed, some experiments cannot be executed in *Stan-*

dardEMF because the simulation runs out of memory. On the contrary, *TemporalEMF* maintains a low memory footprint, using less than 40 MB consistently.

Regarding the disk usage, *TemporalEMF* requires ∼2 times more storage than *StandardEMF*. However, *TemporalEMF* can take advantage of the distributed HBase infrastructure, thus allowing models to grow beyond the storage available in a single machine.

RQ3 — Table 3 shows the results for the experiments to evaluate *Reproduction Cost*. We executed **Q1**–**Q4** ondifferent models for both *StandardEMF* and *TemporalEMF*. As expected *StandardEMF* outperforms *TemporalEMF* since all needed information is already in memory. However, *StandardEMF* is not scalable, and fails when models start growing or when memory is limited. On the contrary, *TemporalEMF* is able to execute all the queries in all the evaluated setups, even when memory is tightly constrained. It is worth noting that some queries are more costly than others (e.g., **Q2** and **Q3** vs **Q1** and **Q4**). In any case, most of the queries can be computed in very few millisecons, and only **Q1** on specially big models takes several seconds (to return hundreds of thousands of elements).

5.3 Threats to Validity

Several internal and external factors may jeopardize the validity of our results. The first ***internal threat*** to validity is about the applied pattern for the *StandardEMF* solution. There are different patterns for keeping temporal information in object-oriented structures (even making use of external databases, thus alleviating the memory consumption). We used a standard pattern, but other well-known patterns may show a different result. The same holds for the formulation of the queries.

There are also ***external threats*** which may jeopardize the generalization of our results. First, we only performed one case study in the domain of model simulation domain. Other domains may show different ratios between the number of static design elements and dynamic runtime elements. Moreover, we did not allow changes to the design element during simulation. Finally, also the employed queries may not represent all possibilities on how to access a temporal model. We aimed to provide heterogeneous queries, such as provenance queries and KPI formulas. However, other queries such as retrieving full model states or a revision graph for the complete model evolution may require different capabilities and may show different runtime results.

6 Related Work

While there is abundant research work on temporal modeling and query languages to for systems data (e.g., consider [15] or [23] for a survey), ours is, as far as we know, the first fine-grained temporal metamodeling infrastructure, enabling the transparent and native tracking (and querying) of system models themselves.

Closest approaches to ours are model versioning tools, focusing on storing models in Version Control Systems (VCS) such as SVN and Git using XMI serializations [2] as well as in database technologies such as relational databases, graph databases, or tuple stores [3]. Traditionally, each version of an evolving model is stored as self-contained model instance together with a timestamp on when the instance as a whole was recorded in the VCS. There is no temporal information at the model element level, and versions are

generated on demand when the designer feels there are enough changes to justify a new version (and not based on the temporal validity of the model). Therefore, reasoning on the history of specific elements with a sufficient degree of precision is barely impossible.

Trying to adapt versioning systems to mimic a temporal metamodeling infrastructure would trigger scalability issues as well. Storing full model states for each version is not efficient. E.g., several approaches use model comparison [25] to extract fine-grained historical data out of different model versions. However, these solutions are extremely costly. Just consider changing one value between two versions. This would result in mostly two identical models which have to be stored and compared. This clearly shows that historical model information is currently not well supported by existing model repositories.

A second group of related work is the family of models@run.time approaches [4]. Models@run.time refers to the runtime adaptation mechanisms that leverage software models to dynamically change the behaviour of the system based on a set of predefined conditions. While these approaches provide a modeling infrastructure to instantiate models, as we do, they do not store the history of those changes and only focus on the current state to steer the system. The only exception is the work by Hartmann at al. [17] which proposes the usage of versioning as we have seen in versioning systems for models. Instead of full models, model elements are versioned. However, the versions have to be explicitly introduced and managed as in the aforementioned versioning systems. We find a similar situation with the group of works on model execution [10,13] that focus on representing complete model states but do not keep track of the evolution of those states unless the designer manually adds some temporal patterns (e.g. the one in the previous section).

7 Conclusion

We have presented *TemporalEMF*, a temporal modeling infrastructure built on top of EMF. With *TemporalEMF*, conceptual schemas are automatically and transparently treated as temporal models and can be subject to temporal queries to retrieve and compare the model contents at different points in time. An extension to the standard EMF APIs allows modelers to easily express such temporal queries. *TemporalEMF* relies on HBase to provide an scalable persistence layer to store all past versions.

As further work, we would like to extend *TemporalEMF* in several directions. At the modeling level, we will predefine some useful temporal patterns to facilitate the definition of temporal queries and operations. At the technology level, we will explore the integration of our temporal infrastructure in other types of NoSQL backends and Web-based modeling environments to expand our potential user base. Finally, we aim to exploit the generated temporal information for a number of learning and predictive tasks to improve the user experience with modeling tools. For instance, we could classify users based on their typical modeling profile and dynamically adapt the tool based on that behaviour.

References

1. Temporal EMF, `http://hdl.handle.net/20.500.12004/1/A/TEMF/001`
2. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. IJWIS 5(3), 271–304 (2009)

3. Barmpis, K., Kolovos, D.S.: Comparative analysis of data persistence technologies for large-scale models. In: Proc. of Extreme Modeling Workshop. pp. 33–38 (2012)
4. Bencomo, N., France, R.B., Cheng, B.H.C., Aßmann, U. (eds.): Models@run.time - Foundations, Applications, and Roadmaps, LNCS, vol. 8378. Springer (2014)
5. 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: Proc. of MODELS. pp. 181–188 (2017)
6. Benelallam, A., Gómez, A., Tisi, M., Cabot, J.: Distributing relational model transformation on MapReduce. J. Syst. Softw. 142, 1 – 20 (2018)
7. Bézivin, J.: On the unification power of models. Softw. Syst. Model. 4(2), 171–188 (2005)
8. Bill, R., Mazak, A., Wimmer, M., Vogel-Heuser, B.: On the need for temporal model repositories. In: Proc. of STAF Workshops. pp. 136–145 (2018)
9. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, 2nd Edition. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers (2017)
10. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G.: Challenges and directions in formalizing the semantics of modeling languages. Comput. Sci. Inf. Syst. 8(2), 225–253 (2011)
11. Cabot, J., Olivé, A., Teniente, E.: Representing Temporal Information in UML. In: Proc. of UML. pp. 44–59 (2003)
12. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. In: Proc. of OSDI. pp. 15–15 (2006)
13. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of UML models: a systematic review of research and practice. Software & Systems Modeling (2018)
14. Gómez, A., Tisi, M., Sunyé, G., Cabot, J.: Map-based transparent persistence for very large models. In: Proc. of FASE. pp. 19–34 (2015)
15. Gregersen, H., Jensen, C.S.: Temporal entity-relationship models - A survey. IEEE Trans. Knowl. Data Eng. 11(3), 464–497 (1999)
16. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15(4), 287–317 (Dec 1983), DOI: 10.1145/289.291
17. Hartmann, T., Fouquet, F., Nain, G., Morin, B., Klein, J., Barais, O., Traon, Y.L.: A native versioning concept to support historized models at runtime. In: Proc. of MODELS. pp. 252–268 (2014)
18. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: EMF profiles: A lightweight extension approach for EMF models. Journal of Object Technology 11(1), 1–29 (2012)
19. Mazak, A., Wimmer, M., Patsuk-Boesch, P.: Reverse engineering of production processes based on markov chains. In: Proc. of CASE. pp. 680–686 (2017)
20. Mazak, A., Wimmer, M.: Towards liquid models: An evolutionary modeling approach. In: Proc. of CBI. pp. 104–112 (2016)
21. OMG: Object Constraint Language (OCL), Version 2.3.1 (January 2012), http://www.omg.org/spec/OCL/2.3.1/
22. Runeson, P., Höst, M.: Guidelines for Conducting and Reporting Case Study Research in Software Engineering. Empirical Software Engineering 14(2), 131–164 (2009)
23. Soden, M., Eichler, H.: Temporal Extensions of OCL Revisited. In: Proc. of ECMFA. pp. 190–205 (2009)
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009), ISBN: 0321331885
25. Stephan, M., Cordy, J.R.: A survey of model comparison approaches and applications. In: Proc. of MODELSWARD. pp. 265–277 (2013)
26. The Apache Software Foundation: Apache HBase (2018), http://hbase.apache.org/