A Native Versioning Concept to Support Historized Models at Runtime

Thomas Hartmann, François Fouquet, Gregory Nain, Brice Morin, Jacques Klein, Olivier Barais, Yves Le Traon

A Native Versioning Concept to Support Historized Models at Runtime

Thomas Hartmann[1], Francois Fouquet[1], Gregory Nain[1], Brice Morin[2], Jacques Klein[1], Olivier Barais[3], and Yves Le Traon[1]

[1] Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg
first.last@uni.lu, http://wwwen.uni.lu/snt
[2] SINTEF ICT Norway
first.last@sintef.no, http://www.sintef.no
[3] IRISA / INRIA Centre Rennes Bretagne-Atlantique, Université de Rennes 1
last@irisa.fr, http://www.irisa.fr

Abstract. Models@run.time provides semantically rich reflection layers enabling intelligent systems to reason about themselves and their surrounding context. Most reasoning processes require not only to explore the current state, but also the past history to take sustainable decisions *e.g.* to avoid oscillating between states. Models@run.time and model-driven engineering in general lack native mechanisms to efficiently support the notion of history, and current approaches usually generate redundant data when versioning models, which reasoners need to navigate. Because of this limitation, models fail in providing suitable and sustainable abstractions to deal with domains relying on history-aware reasoning. This paper tackles this issue by considering history as a native concept for modeling foundations. Integrated, in conjunction with lazy load/storage techniques, into the Kevoree Modeling Framework, we demonstrate onto a smart grid case study, that this mechanisms enable a sustainable reasoning about massive historized models.

Keywords: Models@run.time, Model-driven engineering, Model versioning, Historized models

1 Introduction

The paradigm of Models@run.time [8], [26] empowers intelligent systems with a model-based abstraction causally connected to their own current state. This abstract self-representation can be used by reasoning processes at runtime. For instance, this enables systems to (i) dynamically explore several adaptation options (models) in order to optimize their state, (ii) select the most appropriate one, and (iii) run a set of verifications of viability on new configurations before finally asking for an actual application. This capability enables the development of safer and more intelligent software systems. However, reasoning on the current state of the system is sometimes not sufficient. Indeed, if the system

only reacts to the current state, it may become unstable, oscillating between two configurations as conflicting events are continuously detected. To avoid this state flapping, it is necessary to consider historical information to compare past versions, detect correlations and take more sustainable and stable adaptation decisions. This scenario and the associated challenges are also illustrated in an industrial context. Creos Luxembourg S.A. is the main energy grid operator in Luxembourg. Our partnership with them is geared at making their electricity grid able to self adapt to evolving contexts (heavy wind or rain, consumption increase) to better manage energy production and consumption. This requires to make predictions on the basis of current and historical data. Here, a linear regression of the average electric load values of the meters in a region, over a certain period of time, has to be computed in order to predict the electric load for this region. This obviously requires access to the model history.

Usually, dynamic modifications operated by intelligent systems at runtime react to small changes in the state (parameter changes; unavailability of a component). These adaptations often enact only few changes to make the system fit better to its new context. Being a slight change in the execution context, or on the system's state, all these changes create successive versions. These changes have to be tracked to keep the history and help reasoners in making decisions.

Unfortunately, Models@run.time in particular and model-driven engineering in general lack native mechanisms to efficiently support the notion of model versioning. Instead, current modeling approaches consider model versioning mainly as an infrastructural topic supporting model management in the sense of version control systems commonly used for textual artefacts like source code [6], [22]. Moreover, current approaches focus more on versioning of meta-models, with a lesser emphasis on runtime/execution model instances. In contrast to this, our versioning approach regards the evolution of models from an application point of view allowing to keep track and use this evolution of domain models (at runtime) at an application level (like *e.g.* Bigtable [12] or temporal databases [25]).

The approach presented in this paper is a general concept to enable versioning of models (as runtime structures) and is not restricted to Models@run.time paradigm, although our approach is very well suited for this paradigm. An efficient support would include (1) an appropriate storage mechanism to store deltas between two model versions, and (2) methods to navigate in the modeling space (as usual), but also to navigate in history (*i.e.* in versions). To overcome this limitation, current implementations usually create their own *ad-hoc* historization solutions that usually come with at least two major drawbacks. First, *ad-hoc* mechanisms make the maintenance complicated and sometimes less efficient than native mechanisms. Secondly, the realization of the models storage is often a simple list of complete models for each change (or periodically), creating either a linear explosion of the memory needed for storage, or a strong limit in the history depth. Moreover, the combination of these two drawbacks makes the navigation in space and versions (models and history) a real nightmare for developers in terms of algorithmic complexity, performance and maintenance.

This paper tackles this issue by including versioning as a native concept directly managed within *each model element*. This inclusion comes with native mechanisms to browse the versions of model elements to enable the navigation in space (model) and history (versions). The paper is structured as follows. Section 2 describes the fundamental ideas and mechanisms of our contribution. Section 3 gives details on how we implemented this idea into the Kevoree Modeling Framework. Based on this implementation, we evaluate our approach in section 4 on a smart grid case study and compare it to classical approaches. Finally, we discuss the related work in section 5 before section 6 concludes.

2 Native Versioning for Models at Runtime

This section describes the concepts and mechanisms, which are necessary to enable (1) the versioning of model elements and (2) the navigation in space (model) and history (versions).

2.1 A *Path* to Reach Elements in the Modeling Space

There are different ways to identify an element within a model: static identity-based matching (unique identifiers), signature-based matching [30], similarity-based matching, and custom language-specific matching [10], [23]. This is needed, for example, to detect changes in a model, and to merge and compare models. To allow model elements to evolve independently and enable an efficient versioning of these elements, we rely in our approach on unique identifiers (UID). We use the *path* of a model element as its unique identifier within a model. Directly inspired by the select operator of relational databases and by XPath [14], the *path* defines a query syntax aligned with the MOF [29] concepts. The navigation through a relationship can be achieved with the following expression: **relationName[IDAttributeValue]**. The **IDAttribute** is one attribute tagged as ID in the meta-model. This expression defines the **PATH** of an element in a MOF relationship. Several expressions can be chained to recursively navigate the nested model elements. Expressions are delimited by a `/`. It is thus possible to build a unique path to a model element, by chaining all sub-path expressions needed to navigate to it from the root model element, via the containment relationships. For instance, let us consider a model that has a collection *vehicles* of *Vehicle*, identified by a plate number. Further, each vehicle has a collection *wheels* of *Wheels* identified by their location (FL:Front Left, etc.). In this case, the path to access the front left wheel of the vehicle "KG673JU" is: *vehicles[KG673JU]/wheels[FL]*. Our definition of UID only relies on domain elements (relationships and attributes) and thus does not require an additional technical identifier. It is important to note that the **IDAttribute** alone does not define uniqueness. Uniqueness is only defined in combination with the reference. Therefore, in the example two wheels named FL can exist but only in two different vehicles.

2.2 Reaching Elements in the Versioning Space

The mechanism of path allows to uniquely identify and efficiently access model elements in the modeling space, but does not consider the notion of version. Since elements in a model usually evolve at different paces, versions shouldn't be considered at the model level but on a model element level. For example, the rim of the front left wheel of vehicle "KG673JU" could have been changed after an accident. Therefore, we could have two versions of the front left wheel, one with the old rim and one with the new one. Using only the path *vehicles[KG673JU]/wheels[FL]* to identify the wheel is not sufficient. To address this new requirement, we introduce a version identifier (VI) in conjunction with the path. This makes it possible to retrieve a model element in a specific version and enables the base navigation in history (space of versions). A timestamp, a number, or a string are examples for valid VIs. A model element version is therefore uniquely identified by its path together with a version identifier. This is shown in figure 1. This concept allows to create an arbitrary number of ver-

Fig. 1. Model element version identifier

sions during the lifetime of an element. Figure 2 shows an exemplary lifecycle of a model element e. The first version (with version identifier v_1) of model element e is created at time t_1 and added to the model. Then e evolves over time and two additional versions (with version identifiers v_2 respectively v_3) are created at t_2 respectively t_3. Finally, at t_4 e is removed from the model and its lifecycle ends. The extension of the concept of path with a version identifier enables basic

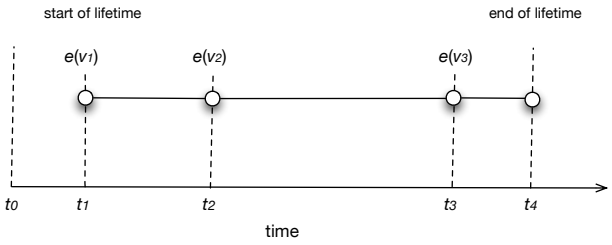

Fig. 2. Lifecycle of a model element

navigation in modeling space and versions, without any impact on the business modeling API. To support this non intrusiveness, and allow for more complex navigation in space and versions, we use the notion of *navigation context*.

2.3 Enhancing Navigation Capabilities with Navigation Context

To ease the navigation in versions, we enrich the modeling API with three basic operations to navigate in versions. These can be called on each model element:

The *shift* operation switches the modeling space to another version. The *previous* operation is a shortcut to retrieve the direct predecessor (in terms of version) of the current model element. The *next* method is similar to the *previous* operation, but retrieves the direct successor of the current model element.

These operations allow to independently select model element versions. Even if this mechanism is powerful, the navigation in such models can rapidly become very complicated. Indeed, a relationship r from an element e_1 to an element e_2 is no longer uniquely defined because of the versioning. Thus, the navigation between model elements can no more rely on relations (in space) only. Figure 3 shows two model elements, e_1 with only one version and e_2 with three versions. The version of e_2, which has to be returned when navigating the relationship r

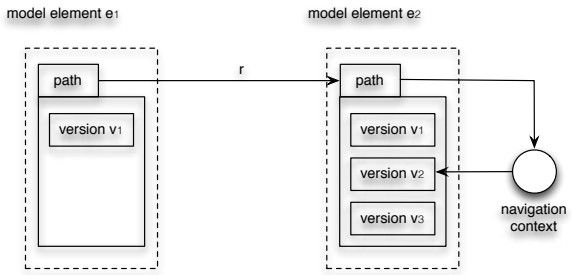

Fig. 3. Different versions of a model element and their relations

is ambiguous. Since each model element can have several versions, a selection policy has to be defined to navigate from one model element to a related. This navigation is thus along two dimensions (space and version). To cope with this problem, we define a *navigation context*.

This navigation context can be either set globally for all model elements, *e.g.* to always return the *latest* or *first* version, or can be set individually for each model element. The navigation context for a model element can also be set to a *specific version*. For example, for the scenario of figure 3 the navigation context for model element e_2 could be set to version v_2. When the model is traversed from e_1 to e_2 using relationship r, version v_2 of e_2 will be returned.

This resolution is transparent and hidden behind methods to navigate in the model. Unlike in previous approaches (*e.g.* relationships in MOF [29]), the navigation function is no longer constant but yields different results depending on the navigation context.

2.4 Leveraging Traces to Store Versions

The storage of model element versions relies on the possibility to get a serialized representation of all attributes and relationships of a model element. For this purpose we use so called *traces*. A trace defines the sequence of atomic actions necessary to construct a model element (or a complete model). Each model element can be transformed into a trace and vice versa [9]. A trace includes all

attribute and relationship information of the model element. Listing 1.1 shows a trace for one model element.

Listing 1.1. Trace of a model element

```
{
"type":"SET","src":"meters[m3]","refname":"consumption","content":"100kWh"
"type":"ADD","src":"meters[m3]","refname":"reachable","content":"hubs[hub2]"
}
```

The listing shows a trace of a model element with an attribute *consumption* and a relationship *reachable*. The value *hubs[hub2]* of relation *reachable* shows how we leverage the path concept in our traces to uniquely identify a model element within a model. We use the JSON [15] format for a lightweight representation and storage of traces.

The version identifier in conjunction with the path (extended path) can be used as *key* and the trace as *value*. This is shown in figure 4. The data can then

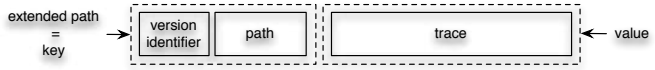

Fig. 4. Storage concept for versioned model elements

be stored using arbitrary back ends *e.g.* key/value stores, relational databases, in RAM (as cache), or even in common version control systems like Git [2].

This section described the foundation of our contribution. To assess its suitability in real cases, we implemented this idea into the KMF project [17].

3 Implementation in KMF

This section presents the implementation of the proposed approach into the Kevoree Modeling Framework. The source is available on the project web page[4].

3.1 The Kevoree Modeling Framework

KMF [17], [18] is an alternative to EMF [11], specifically designed to support the Models@run.time paradigm in terms of memory usage and runtime performance. The code generator of KMF generates a modeling API from an Ecore metamodel and offers several features, *e.g.* (1) event-based listeners, (2) management of persistence, (3) different serialization formats (XMI [28] and JSON [15]), and more recently, (4) the option to compile for Java [19] and JavaScript [16].

3.2 Unique IDs and Model Elements Paths in KMF

As described in section 2, our contribution assumes the availability of an attribute, which is able to uniquely identify model elements within relationships. To enforce this, KMF generates a method *getID()*. If one or several attributes are

[4] http://kevoree.org/kmf

tagged as IDs in the meta-model, the *getID()* method returns a concatenation of the ID attributes' values, ordered alphabetically on the attributes' names. If no ID attribute is specified, KMF adds one and the generated API automatically injects a UID value at creation, though developers are strongly encouraged to provide a more readable, domain-specific value.

In addition, we also assume the uniqueness of the container for any model element. This property is actually ensured by EMF, and KMF also complies to this convention: apart from the root element that is not contained, every model element has to be present once in a containment collection within the entire model [29]. Then, the unique identifier ensures the uniqueness of model elements in the containment collection. By chaining these pieces of information from the root, KMF can create a path that uniquely identifies each model element in the scope of the model. This defines the semantics to navigate in the model along the space dimension.

As presented in the contribution section, the activation of the versioning of model elements implies an extension of this path mechanism to enable the localization of a model element in both version and modeling dimensions. If the conjunction of the version identifier (VI) to the path is a simple idea, its interpretation to resolve a real model element is more intricate. Moreover, it is important to not pollute the modeling space navigation with versioning concerns. Therefore, we introduce a navigation context that is used to precisely drive both the navigation in versions and support the resolution mechanisms of modeling elements (used by the path resolver). This navigation context is implemented by a special object given to the factory of the generated API. This object seamlessly determines which version should be resolved while the model is traversed.

3.3 Traces and Load/Save Mechanisms

Loading and saving model versions can be efficiently managed by big data-like tools, such as key-value stores. The extended path (path and version ID) of a model element is used as key, and the serialized trace of this elements as value. We provide an expandable datastore interface and several implementations of NoSQL storages. Google LevelDB [5] is used by default. It is optimized for an efficient handling of big data, can be easily embedded into applications, and most importantly, it has proved to be very fast for our purpose (see section 4). The data storage implementation itself is not part of our contribution, instead we intend to provide an efficient layer for versioning of model elements on top of already existing storage technologies. As history data can quickly become very big (millions of elements), they may no longer fit completely into memory. We thus implement a lazy loading [1] mechanism, which leverages our serialization strategy and our notion of path. Attributes and relationships are now only loaded when they are accessed (read or written). Until this happens, we use proxies [1] containing only the path and version identifier, to minimize memory usage. This has been achieved by extending KMF so that relationships are dynamically resolved when the model is traversed. It is important to note that our proxy mechanism works at the model element level rather than at the

model level. It first must be determined which version of a related model element must be retrieved. This depends on the navigation context. After this, the actual model element version can be loaded from storage. Load (*get*) and save (*put*) operations are very efficient using extended paths as keys to uniquely identify model element versions. Lazy loading a model element in a specific version requires just one *get* operation from the datastore. This allows to manage models, including histories of arbitrary size, efficiently and it hides the complexity of resolving and navigating versioned data behind a modeling API.

3.4 Navigation Mechanisms

Modeling approaches use meta-model definitions to derive domain specific APIs. Following this idea, our implementation generates an extended API that in addition provides operations to manipulate and navigate the history of model elements. It is illustrated here on a simplified smart grid meta-model definition that consists of a smart meter with an attribute for electric load and a relationship to the reachable surrounding meters. The API provides functions to create, delete, store, load, and navigate model element versions. In addition the API can be used to specify the navigation context on which elements should be resolved while navigating the model. Listing 1.2 shows Java code that uses a Context **ctx** (abstraction to manipulate model element versions) demonstrating some of these operations. In the first part of the listing below, the modeling API is used to create and manipulate a version v_1 of a smart meter element e_1. In the second, the navigation context is defined so that element e_1 is resolved to version v_1 and e_2 to v_2.

Listing 1.2. Usage of the modeling API

```
// creating and manipulating model element versions
e1_1 = ctx.createSmartMeter("e1","v1");
e1_1.setElectricLoad(712.99);
e1_1.addReachables(ctx.load("e2"));
e1_2 = m1.shift("v2");
e1_2.setElectricLoad(1024.4);

// definition of the navigation context
ctx.navDef("e1","v1);
ctx.navDef("e2","v2");
r_e1 = ctx.load("e1");
assert(r_e1.getElectricLoad()==712.99);
r_e2 = r_e1.getReachables().get(0);
assert(r_e2.getVersion()=="v2")
```

4 Evaluation

The native support of versioning at the level of model elements enables the construction of domain specific models, aware of history that, for instance, can empower reasoning processes. To evaluate this claim, this section analyses the impact of using our approach on an industrial case study, which is provided by Creos Luxembourg. In a nutshell, with this case study we evaluate the performance of a model-based reasoning engine that aggregates and navigates smart

grid state information to take corrective actions, like shutting down a windmill in case of overproduction. This reasoning is based on a domain model, which is periodically filled with live data from smart meters and sensors. In this context, our approach is used to store and track the history of the smart grid sate and smart-grid elements' values. A new version of a model element is created each time this model element is updated.

The validation is based on three key performance indicators (KPI): (1) evolution of time and memory required to update a value in the model, (2) gain on time for standard navigation operations in the model (*e.g.* for a reasoning process), and (3) impact on the size required for the persistence of history-aware models. For each KPI, we compare our approach with the classic model sampling strategy taking a snapshot of the entire model for each modification (or periodically). The measured memory value for KPI-1 is main memory (RAM), for KPI-2 disk space. The time measured is the time required to complete the process (depending on the KPI). All experiments are conducted on an Intel core i7 CPU with 16GB RAM and an SSD disk. The full sampling approach and our approach both use a Google LevelDB database for storage and run on the Java Virtual Machine 8. We start our evaluation with a description of the meta-model used in the case study.

4.1 Smart Grid Meta-Model

The smart grid is an emerging infrastructure leveraging modern information and communication technology (ICT) to modernize today's electricity grid. Figure 5 shows an excerpt of the smart gird meta-model that we designed together with our industrial partner Creos Luxembourg. It describes the concepts required to model and store an abstraction of the smart grid infrastructure currently deployed in Luxembourg. We use this meta-model to evaluate all KPIs. This meta-model is of central importance for the following evaluation. Smart meters, concentrators, and their topology allow to reason about communication systems and messages exchanged, while electric segments and measured consumption data are used to reason about consumption/production. Smart meters installed at customers sites continuously measure consumption information and propagate it through network communication links. Concentrators then push these data into the smart grid domain model.

4.2 KPI-1: Impact on Model Updates (CRUD)

To evaluate the impact of our approach on model update operations, we analyse modifications of two magnitudes: (1) a large update (LU) that consists in the creation of a new concentrator and a smart meter subtree (1000 units) and (2) a minor update (MU) that consists in updating the consumption value measured for a specific smart meter already present in the domain model. The size of each update is constant and we vary the size of the domain model and the history (number of versions) of measured values, by repeating the addition of model elements. We grow the domain model from 0 to 100.000 elements, which

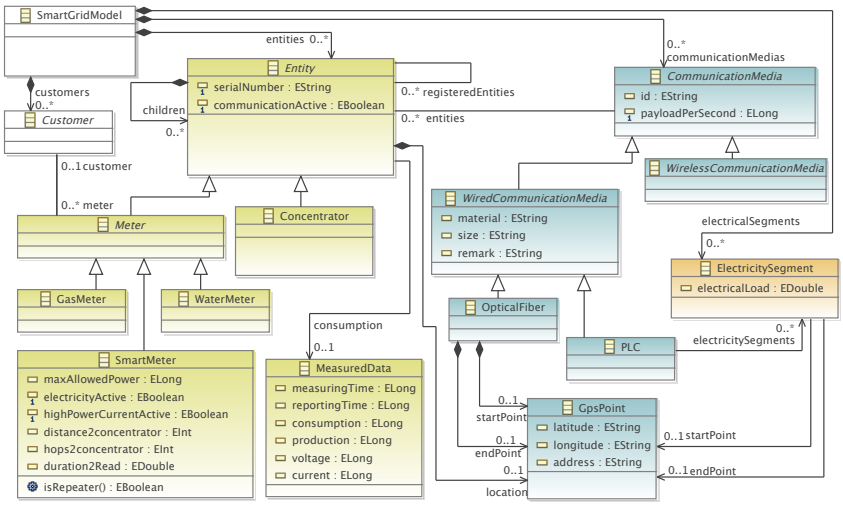

Fig. 5. Smart grid meta-model [4]

approximately corresponds to the actual size of our Luxembourg smart grid model. The results of KPI-1, in term of heap memory and time, are depicted in figure 6 and figure 7. The full sampling strategy is presented on the left, our approach on the right.

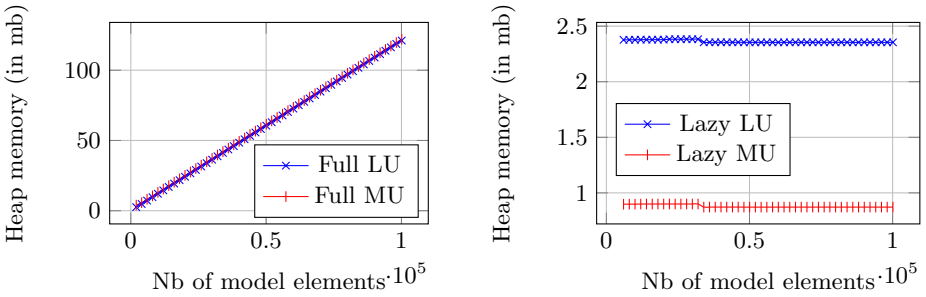

Fig. 6. Memory impact of model manipulation with full and lazy sampling strategies

Let us first consider **memory**. The full sampling strategy depends on the size of the model, as reflected by the linear progression of the heap memory size required to insert fixed size updates. In contrary, our approach results in two flat curves for LU and MU updates, showing that the memory required depends only on the size of the update, not on the model size. This is verified by the fact that minor modifications (MU) require less memory than LU and both are constant and under 2.5MB while the full sampling requires up to 100MB.

Let us now consider **time**. Again, the time to insert new elements with full sampling is related to the size of the model, but nearly constant with our solution.

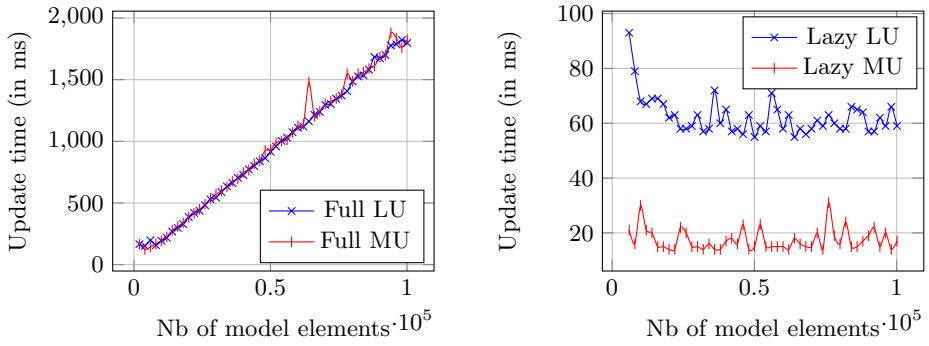

Fig. 7. Time and memory impact of model manipulation with full sampling strategy

Also, the time reduction is more important for MU than LU, confirming that our approach reduces the necessary time to modify elements. KPI-1 demonstrates that even in the worst case scenario, where all elements evolve at the same pace, our approach offers a major improvement for CRUD operations (factor of 33 for time and between 50 to 100 for memory).

Finally, let us consider **batch insertions**. To validate this result, we additionally performed a batch insert operation in full sampling and with our solution. This batch consists of 10.000 historical values for each meter, resulting in a model of 1 million elements. We obtain as result **267s** to insert with the full sampling strategy and **16s** for our approach. Even in this worst case, we still have an improvement of a factor **17** for the insertion time.

4.3 KPI-2: Impact on Time Required for Exploration in Versions and Reasoning Process

For this evaluation, we consider an already existing smart grid model containing measurement values. The goal is to measure the gain (or loss) of time needed to execute a complex computation on this history. We run several prediction algorithms on the model, which correlate historical data to evaluate the current state of the grid and, for example, throw an alert in case of an overconsumption.

We define two kind of predictions for the smart grid, at two different scales, resulting in 4 different reasoning strategies: (1) small deep prediction (SDP), (2) small wide prediction (SWP), (3) large deep prediction (LDP), and (4) large wide prediction (LWP). Wide prediction means that the strategy uses a correlation of data from neighbour meters, to predict the future consumption. In a deep prediction, the strategy leverages the history of the customer to predict its consumption habits. Both approaches perform a linear regression to predict the future consumption using two scales: large (100 meters) and small (10). Results and time reduction factors are presented in the table 1. The gain factor of our approach, compared to full sampling, is defined as $Factor = (Full\ Sampling\ time\ /\ Native\ Versioning\ time)$. The gain factor is between **557** and **1330**, reducing the processing time from minutes to seconds.

Table 1. Reasoning time in ms for consumption predictions

Type	**SDP**	**SWP**	**LDP**	**LWP**
Full	1147.75 ms	1131.13 ms	192271.19 ms	188985.69 ms
Lazy	2.06 ms	0.85 ms	189.03 ms	160.03 ms
Factor	557	1330	1017	1180

Although, we perform the computation for only one point of the grid, it has to be multiplied by the number of smart meters to evaluate in a district. Now, the gain highlighted here has already allowed to drop the time required to analyse the entire grid, from hours of computation to seconds. Beyond Models@run.time usage, this enables reasoning processes to react in near real-time (milliseconds to seconds), which is required for smart grid protection reactions.

4.4 KPI-3: Versioning Storage Strategy Overhead Evaluation

In this section, we study and evaluate the overhead induced by our approach, compared to the classic full model sampling strategy. Our goal is to detect the percentage of modifications of a model, in a single version, above which the full sampling approach creates less overhead. In other words, which percentage of modifications makes the overhead of our solution a disadvantage in terms of storage, despite its navigation gains are still valid.

For this evaluation, we load an existing model (containing 100 smart meters), update the consumption value of several meters, serialize it again and store the size. By varying the percentage of meters updated per version (period), we can compare the size of the storage required for the diff with our approach, and the full sampling. To ensure a fair comparison we use a compact JSON serialisation format for both strategies. Results are depicted in Figure 8. The full sampling

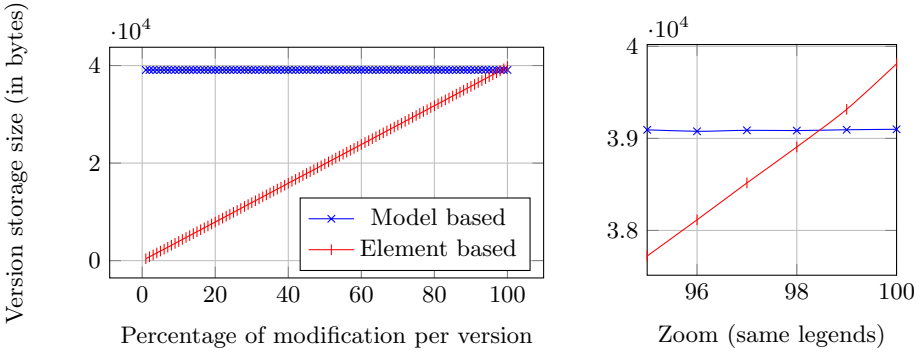

Fig. 8. Impact on the storage required to save versions

mechanism implies 39.1Kb of storage per model, regardless of the percentage of modifications. This is a serious overhead for small modifications. Our strategy requires a variable amount of memory, from 393 bytes for 1% of change to 39.8Kb for 100% (complete change of the model). Also, linear augmentation of changes

in the model with our approach creates a linear augmentation of the storage. This confirms that our independent storage strategy for each model element has no hidden side effect.

Our storage offers a reduction of **99.5**% for 1% of change, but an increase of only 1.02% for 100% of modifications. This means that, up to **98.5%** of modifications of a model, our approach needs less memory than full sampling. Also, the overhead of 1.02% for full model change storage has to be related to the features enabled by this overhead (navigation, insertion time gains, comparison time gains). In this context, we consider the overhead acceptable.

Beside runtime usage improvements, this validation proves that we can offer nearly constant memory and time consumption for model-based contexts, which allows to face potentially massive history. These improvements are mainly due to lazy-load and efficient path mechanisms, which by construction reduce the floating memory window to read and write model elements. Similarly to how NoSQL databases scale, we carefully reuse the modeling hierarchy concept to fit with the datastore organization that offers the best performance, which explains this very considerable gain. Finally, this validation demonstrates that our approach is suitable for the manipulation of massive historical model.

5 Related Work

Considering versioning (or time) as a crosscutting concern of data modeling has been discussed for a long time, especially in database communities. In [13] Clifford *et al.* provide a formal semantic for historical databases and an intentional logic. Rose and Segev [31] incorporate temporal structures in the data model itself, rather than at the application level, by extending the entity-relationship data model into a temporal, object-oriented one. In addition they introduce a temporal query language for the model. Ariav [7] also introduces a temporally-oriented data model (as a restricted superset of the relational model) and provides a SQL-like query language for storing and retrieving temporal data. The works of Mahmood *et al.* [25] and Segev and Shoshani [33] go into a similar direction. The later also investigate the semantics of temporal data and corresponding operators independently from a specific data model in [32]. In a newer work [12], Google embeds versioning at the core of their BigTable implementation by allowing each cell in a BigTable to contain multiple versions of the same data (at different timestamps).

The necessity to store and reason about versioned data has also been discussed in the area of the Semantic Web and its languages, like RDF [24] and OWL [35]. For example, Motik [27] presents a logic-based approach for representing versions in RDF and OWL.

Recently, the need to efficiently version models has been explored in the domain of model-driven engineering. However, model versioning has been mainly considered so far as an infrastructural issue in the sense that models are artifacts that can evolve and must be managed in a similar manner to textual artifacts like source code. Kerstin Altmanninger *et al.* [6] analyze the challenges coming

along with model merging and derive a categorization of typical changes and resulting conflicts. Building on this, they provide a set of test cases which they apply on state-of-the-art versioning systems. Koegel and Helming [22] take a similar direction with their EMFStore model repository. Their work focuses on how to commit and update changes and how to perform a merge on a model. Brosch *et al.* [10] also consider model versioning as a way to enable efficient team-based development of models. They provide an introduction to the foundations of model versioning, the underlying technologies for processing models and their evolution, as well as the state of the art. Taentzer *et al.* [34] present an approach that, in contrast to text-based versioning systems, takes model structures and their changes over time into account. In their approach, they consider models as graphs and focus on two different kinds of conflict detection, operation-based conflicts between different graph modifications and the check for state-based conflicts' on merged graph modifications. These works consider versioning at a model level rather than at a model element level. Moreover, these approaches focus on versioning of meta-models whereas our work focuses on versioning of runtime/execution models. Our approach enables not only to version a complete model, but considers versioning and history as native mechanisms for any model element. Moreover, versioning in the modeling domain is usually considered from a design / architecture / infrastructural point of view, and models are versioned as source code files would be. In contrast to this, our versioning approach regards the evolution of model elements from an application point of view (*e.g.* Bigtable [12] or temporal databases [25]). It allows to keep track of the evolution of domain model elements —their history— and use this history efficiently on an application level.

Most of the above mentioned work address storing and querying of versioned data but largely ignores the handling of versioning at an application level. However, many reasoning processes require to explore simultaneously the current state and past history to detect situations like a system state flapping. Our approach proposes to consider model versioning and history as native mechanisms for modeling foundations. We not only efficiently store historical data (what is done in other works before), but we propose a way to seamlessly use and navigate in historized models. Also, we do not extend a specific data model (*e.g.* the relational data model or object-oriented one) but use model-driven engineering techniques to integrate versioning as a crosscutting property of any model element. We aim at providing a natural and seamless navigation into the history of model element versions. Just like version control systems, *e.g.* Git [2], we only store incremental changes rather than snapshots of a complete model.

6 Conclusion

The use of models to organize and store dynamic data suffers from the lack of native mechanism to handle the history of data in modeling foundations. Meanwhile, the recent evolutions of model-driven engineering, and more precisely, the emergence of the Models@run.time paradigm spreads the use of models to

support reasoning at runtime. Therefore, the need for efficient mechanisms to store and navigate the history of model element values (*a.k.a.* dynamic data) has strongly increased. The contribution presented in this paper aims at addressing this need, adding a version identifier as a first-class feature crosscutting any model element. This approach, coupled with the notion of trace and lazy load/save techniques, allows model elements to be versioned independently from each other without the need to version a complete model. Moreover, this paper describes the navigation methods introduced on each model element to enable the basic navigation in versions. Finally, we defined a navigation context to simplify and improve the performances of navigation between model elements coming from heterogeneous (different) versions.

We evaluate the added value of this work on a case study from the smart grid domain, defined with an industrial partner. The validation relies on an implementation of the approach into the Kevoree Modeling Framework. This evaluation shows the efficiency of the storage and navigation mechanisms compared to full sampling and *ad-hoc* navigation techniques. It also demonstrates that in the worst case (*i.e.* when all model elements are modified at the same pace) the storage overhead is negligible (1.02%), while our navigation mechanism still offer constant performances. Even if the evaluation has been run in use cases linked to the Models@run.time paradigm, we are convinced that this approach can also be used in any kind of applications using versioned data. For example, we use a derivation of this approach to enable what we call *time-distorted* context representations to manage a huge amount of temporal data in runtime reasoning processes [21]. This has been proven especially useful in the context of reactive security for smart grids [20].

In future work we plan to: (i) integrate a declarative query language on top of our approach to improve the selection of model element versions, instead of relying only on the three basic operations *shift*, *previous*, and *next*, (ii) use distributed data stores like HyperDex DB [3], and (iii) define reusable patterns of version navigation to tame the complexity of reasoning process development.

Acknowledgments

The research leading to this publication is supported by the National Research Fund Luxembourg (grant 6816126) and Creos Luxembourg S.A. under the SnT-Creos partnership program.

References

1. CDO eclipsedia. `http://wiki.eclipse.org/CDO`. Accessed: 2014-02-01.
2. Git. `http://git-scm.com/`.
3. HyperLevelDB Performance Benchmarks. `http://hyperdex.org/performance/leveldb/`. Accessed: 2014-02-01.
4. KMF Samples, MoDELS14. `https://github.com/kevoree/kmf-samples/models14`. Accessed: 2014-03-15.
5. leveldb a fast and lightweight key/value database library by google. `https://code.google.com/p/leveldb/`. Accessed: 2014-02-10.
6. Kerstin Altmanninger, Petra Kaufmann, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why model versioning research is needed!? an experience report. In *Proceedings of the Joint MoDSE-MC-CM 2009 Workshop*, 2009.
7. Gad Ariav. A temporally oriented data model. *ACM Trans. Database Syst.*, 11(4):499–527, December 1986.
8. G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.
9. Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 511–520, New York, NY, USA, 2008. ACM.
10. Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In *Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*, SFM'12, pages 336–398. Springer-Verlag, 2012.
11. Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework : A Developer's Guide.* 2003.
12. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.
13. James Clifford and David Scott Warren. Formal semantics for time in databases. In *XP2 Workshop on Relational Database Theory*, 1981.
14. World Wide Web Consortium. Xml path language (xpath) 2.0 (second edition). Technical report, World Wide Web Consortium, 2010.
15. Douglas Crockford. The application/json media type for javascript object notation (json). RFC 4627, IETF, 7 2006.
16. ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1 edition, June 2011.
17. F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J. Jézéquel. An eclipse modelling framework alternative to meet the models@runtime requirements. In *MoDELS*. Springer, 2012.
18. Fouquet Francois, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use. Rapport de recherche TR-SnT-2014-11, May 2014. ISBN 978-2-87971-131-7.

19. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. California, USA, java se 7 edition, February 2012.
20. T. Hartmann, F. Fouquet, J. Klein, G. Nain, and Y. Le Traon. Reactive security for smart grids using models@run.time-based simulation and reasoning. In *Proceedings of the Second Open EIT ICT Labs Workshop on Smart Grid Security (SmartGridSec14)*, 2014.
21. T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, and Y. Le Traon. Reasoning at runtime using time-distorted contexts: A models@run.time based approach. In *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2014.
22. Maximilian Koegel and Jonas Helming. Emfstore: a model repository for emf models. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE (2)*, pages 307–308. ACM, 2010.
23. Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 1–6. IEEE Computer Society, 2009.
24. Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, W3C, February 1999.
25. Nadeem Mahmood, Aqil Burney, and Kamran Ahsan. A logical temporal relational data model. *CoRR*, 2010.
26. B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.
27. Boris Motik. Representing and querying validity time in rdf and owl: A logic-based approach. In *Proc. 9th Int. Semantic Web Conf. The Semantic Web - Volume Part I*, ISWC'10, pages 550–565, 2010.
28. OMG. *XML Metadata Interchange (XMI)*. OMG, 2007.
29. OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. Technical report, Object Management Group, August 2011.
30. R. Reddy, R. France, S. Ghosh, F. Fleurey, and B. Baudry. Model composition - a signature-based approach. In *Aspect Oriented Modeling (AOM) workshop held in conjunction with MODELS/UML 2005 conference, Montego Bay, Jamaica*, 2005.
31. Ellen Rose and Arie Segev. Toodm - a temporal object-oriented data model with temporal constraints. In Toby J. Teorey, editor, *ER*, 1991.
32. Arie Segev and Arie Shoshani. Logical modeling of temporal data. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, New York, NY, USA, 1987.
33. Arie Segev and Arie Shoshani. The representation of a temporal data model in the relational environment. In *SSDBM*, 1988.
34. Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software and System Modeling*, 13(1):239–272, 2014.
35. World Wide Web Consortium W3C. Owl 2 web ontology language. structural specification and functional-style syntax. Technical report, 2009.