Change-based Persistence and Its Loading Optimisation

ABSTRACT

In this paper, we propose a change-based approach for persisting models conforming to object-oriented (MOF/Ecore) metamodelling architectures. We demonstrate how change-based persistence enables high-performance incremental model processing (e.g. transformation, validation) by reducing the cost of identifying fine-grained changes in evolving models. We illustrate a language-independent change-based model representation format and an optimised model loading algorithm that seeks to avoid replaying changes that have no impact on the eventual state of the model. We report on benchmarks that compare model loading and saving performance of the proposed change-based representation against the standard state-based representation format (XMI). Our results show considerable savings in terms of persisting and identifying changes made to models at an increased – but linear – model loading cost.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; *Software performance*; Software architectures;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference Format:

. 2018. Change-based Persistence and Its Loading Optimisation. In *Proceedings of International Conference on Software Engineering, Gothenburg, Sweden, May-June 2018 (ICSE'18),* 10 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

To reap the benefits of Model-Based Software Engineering in the context of complex and large systems, the ability to process large models in an incremental fashion as they evolve is essential. Existing incremental model processing techniques only yield limited performance benefits due to imprecise and slow model change detection capabilities or are only limited to a single-developer environment, which is not applicable for real-world software development projects.

Instead of persisting snapshots of states of models, we propose turning models inside out and persisting their change history. The proposed approach has the potential to deliver step-change performance benefits in incremental model processing, as well as a wide range of other benefits and novel capabilities. While persisting changes to large models is expected to be much faster and resource-efficient compared to state-based approaches, loading models into memory by naively replaying the entire change history is expected to have a significant overhead. To address this challenge, we propose algorithms and data structures that will reduce the cost of change-based model loading (e.g. by recording and ignoring events – events that are later overridden or cancelled out by other events), extending the foundational work of Yohannis et al. [8].

The rest of the paper is structured as follows. Section 2 presents briefly the concept of change-based persistence. Section 3 provides an overview of existing related work. Section 4 overviews our proposed approach. The algorithms to optimise the loading time of change-based persistence, as well as its loading time test, are presented in Sect. 5 and Sect. 6 respectively. Section 7 presents the limitation of our approach and Sect. 8 concludes this paper.

2 CHANGE-BASED PERSISTENCE

A change-based model or change-based persistence (CBP) is a concept of model that records incremental changes – incrementality – of a model and uses the records for a variety of purposes, such as model analytics, optimising execution of model operations (e.g. validation, transformation), or generating different versions of a model based on its timeline. The incremental changes can be presented as, but not limited to, a collection of events arises from operations applied to a model during its construction.

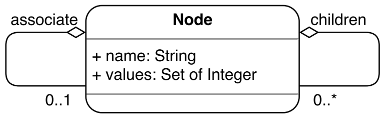

Figure 1: A class diagram of a simple tree modelling language.

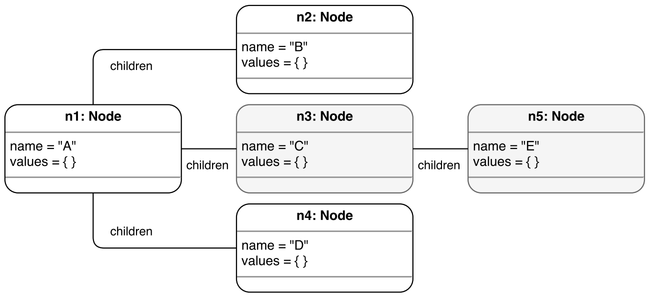

Figure 2: A tree model – node *n3* and *n5* are deleted from a model that also contains nodes *n1*, *n2*, *n3*, and *n4*.

To help us explain the concept of incrementality and our approach to optimise its loading time, we use a simple tree modelling language (Fig. 1 shows its class diagram) to create examples of tree models to support our explanation. Essentially, the model consists of nodes in which every node can have one to many, containment references[1] with other nodes (*children* relationship). A node can can also have a have a single containment reference, *associate* relationship, with another node. Every node has two attributes, *name* and values. Attribute *name* is a type of String and attribute *values* contains a set of Integer.

As the first, main example for this paper, we create a tree model as shown in Fig. 2 using the modelling language in Fig. 1. Initially, the model consists of five nodes *n1*, *n2*, *n3*, *n4*, and *n5*. Node *n1* contains three direct nodes *n2*, *n3*, and *n4*, and node *n3* contains node n5. Later, nodes *n3* are deleted from the model, which means node *n5* is also deleted since node *n5* is contained by node *e3* (deleted nodes are coded with grey colour). After every modification of the model, the model needs to be (1) validated against a domain-specific constraint (a node can only have 3 or less children), and (2) transformed into a number of HTML files through a model-to-text transformation. Each file contains the name of the node, and the names of its direct children.

Before nodes *n3* and *n5* are deleted from the model, when the validation constraint is evaluated against the initial model (Fig. 2), it verifies that each of the five nodes has three or less children, and the model-to-text transformation then produces five HTML files that correspond to the nodes in the model. The model is then updated by deleting nodes *n3* and *n5* from the model, which also node remove node *n3* from node *n1*.

A non-incremental model validation engine, after the removal of nodes *n3* and *n5*, would treat the model of Fig. 2 as a new model and will evaluate the constraint above against every node in the model. An incremental model validation engine, on the other hand, would identify that the previously established satisfaction of the constraint for nodes *n2* and *n4* are not affected by the changes made, and would only re-evaluate the constraint for nodes *n1* instead.

Similarly, a non-incremental model-to-text transformation would generate and overwrite all output text files from scratch. On the contrary, an incremental model-to-text transformation, would identify that it only needs to delete the HTML files of nodes *n3* and *n5* since they have been removed and generate a new HTML file for node *n1* as the number of its children has changed – but not the text files of nodes *n2* and *n4*, as these cannot have been affected by the changes made to the model.

While the overhead of executing transformations and validation constraints on small models like the one in Fig. 2 is not significant, non-incremental execution can become a significant bottleneck for large evolving models, especially when a development cycle is closer to its end, when developers tend to perform many small changes to fine-tune systems. Thus, a sophisticated impact analysis is needed to reduce the amount of code regeneration in order to keep the overhead minimum [7].

3 RELATED WORK

There has been some work related to change-based persistence. In the pioneering work of Egyed [1], in order to achieve incremental re-execution of (deterministic) queries on structured models, an execution engine needs to (1) record model element property accesses during the initial execution of the queries, (2) identify new and deleted elements and modified model element properties in the new version of the model, and (3) combine the information collected in the steps one and two to identify the subset of (potentially) affected rules/queries/templates that need to be re-executed. Egyed has shown that the property-access recording approach is applicable to queries of arbitrary complexity, as long as they are deterministic. More recent work has shown that variants of this approach can be used to achieve incrementality in a wide range of model processing operations, including model-to-model transformation [2], model-to-text transformation [5], model validation, and pattern matching [6]—as long as changes to models can be precisely identified (step 2 above).

There are two approaches in the literature for identifying changes in models in order to enable incremental re-execution of model processing operations, emphmodel differencing and *notifications*. *Model differencing* approach eliminates the coupling between modelling tools and incremental execution engines. Instead of depending on live notifications, in this approach the developer in charge of automated model processing, needs to have access to a copy of the last version of the model that the model processing program (e.g. the model-to-text transformation) was executed upon, so that it can be compared against the current version of the model (e.g. using a model-differencing framework such as SiDiff [3] or EMFCompare[2]) and the delta can be computed on demand. The main advantage of this approach is that it works well in a collaborative development environment where typically developers have distinct roles and responsibilities. On the downside, model comparison and differencing are computationally expensive and memory-greedy (both versions of the model need to be loaded into memory before they can be compared), thus largely undermining the time and resource saving potentials of incremental re-execution. This approach is adopted by the Xpand model-to-text transformation language. According to the developers of the language, using this approach, a speed-up of only around 50% is observed compared to non-incremental transformation[3], which is consistent with our experience from using Xpand.

In *notification* approach, the incremental execution engine needs to hook into the notification facilities provided by the modelling tool through which the developer edits the model, so that the engine can directly receive notifications as soon as changes happen (e.g. a node (*n5*) has been deleted, the name property of node *n5* has been changed to "E"). This is an approach taken by the IncQuery incremental pattern matching framework [6] and the ReactiveATL incremental model-to-model transformation engine [5]. The main advantage of this approach is that precise and fine-grained change notifications are provided for free by the modelling tool (and thus do not need to be computed by the execution engine—which as discussed below can be expensive and inefficient). On the downside, this approach is a poor fit for collaborative development settings where modelling and automated model processing activities are performed by different members of the team.

In summary, incremental model processing currently delivers significant performance benefits only in a single-developer environment where the modeller is also responsible for performing

[1]Containment reference is a composite relationship which an object holds a reference to another object. Deleting the former deletes the the latter as well.

[2]https://www.eclipse.org/emf/compare/
[3]http://wiki.eclipse.org/Xpand/New_And_Noteworthy#Incremental_Generation

all the (incremental) model processing operations. As a result, in collaborative development environments, developers need to either forgo incremental model processing altogether or to work around this limitation by manually steering model processing programs to process only subsets of their models, which is cumbersome and error prone.

4 PROPOSED APPROACH

The ambition of this research is to enable high-performance incremental model management in collaborative software development environments by challenging one of the fundamental assumptions of contemporary modelling frameworks and tools: as opposed to persisting snapshots of the state of models (which is what virtually all modelling tools and frameworks currently do), we propose turning models inside out and persisting their change history instead.

Listing 1: State-based representation of the model of Figure 2 after removal of node *n5* in (simplified) XMI.

```
<Node xmi:id="n1" name="A">
  <children xmi:id="n2" name="B"/>
  <children xmi:id="n3" name="C"/>
  <children xmi:id="n4" name="D"/>
</Node>
```

To illustrate the proposed approach, List. 1 shows a state-based representation of the model of Fig. 2 after the removal of node *n5* in (simplified) XMI, and List. 2 shows the proposed equivalent change-based representation of the same model. Instead of a snapshot of the state of the model, the representation of List. 2 captures the complete sequence of change events (create/set/unset/add/remove/move/delete) that were performed on the model since its creation. Replaying these changes produces the same state as the one captured in List. 1, so the proposed representation carries at least as much information as the state-based representation.

Listing 2: Change-based representation of the model of Figure 2 after removal of node *n5*.

```
1   session s1
2   create n1 of Node
3   set name of n1 with "A"      //m5.name="A"
4   create n2 of Node
5   set name of n2 with "B"      //m5.name="B"
6   create n3 of Node
7   set name of n3 with "C"      //m5.name="C"
8   create n4 of Node
9   set name of n4 with "D"      //m5.name="D"
10  create n5 of Node
11  set name of n5 with "E"      //m5.name="E"
12  add n2 to n1.children        //n1.children={n2}
13  add n3 to n1.children        //n1.children={n2,n3}
14  add n4 to n1.children        //n1.children={n2,n3,n4}
15  add n5 to n3.children        //n3.children={n5}
16  session s2
17  remove n5 from n3.children   //n3.children={}
18  delete n5
19  remove n3 from n1.children   //n3.children={n2,n4}
20  delete n3
```

Such a representation is particularly suitable for incremental model processing. For example, if the model-to-text transformation discussed above "remembers" that in its previous invocation it had processed up to editing session *s1* of the model, it can readily identify the changes that have been made to the model since then instead of having to rediscover them through (expensive) state-based model differencing. For example, in in session *s2* (lines 16-20), nodes *n3* and *n5* are deleted from the model and deletion of nodes n3 changes the number of nodes (children) that belongs to node n1. Therefore, previous generated HTML files then can be removed (for nodes *n3* and *n5*) or regerated (for node *n1*).

From CBP in Lst. 2, we can tell that to produce model as in Lst. 1, the model is constructed in two consecutive sessions (lines 1 and 16). In the first session (*s1*), node *n1* is created and its name attribute is assigned with a string value (lines 2 and 3). The same operations also are also applied to other nodes, from node *n2* to *n5*, sequentially (lines 4-11). After that, nodes *n2*, *n3*, and *n4* are added to node *n1* as its children (lines 12-14) whilst node *n5* is added to node *n3* (line 15). In the second session (*s2*), node *n5* is removed from *n3* and then deleted (lines 17-18), followed by the removal of *n3* from *n1* and its deletion (lines 19-20).

4.1 Notifications and Events

To enable change-based persistence, relevant consecutive operations or events generated need to be captured and then can be transformed to produce change-based representation such as the one showed in List. 2. Existing technologies have already provided notification facilities that enable developers to record such operations and events. For example, Eclipse Modelling Framework has *Notification*[4] class and *notifyChanged*[5] method that can be used to identify change events in a model.

Algorithm 1: An algorithm to capture an event in a change notification method.

```
input : an object of Notification $notification$, an value of
        Integer $lineNumber$, a list of Event $eventList$, an
        object of ModelHistory $modelHistory$
begin
    if $getEventType(notification) = Event.CREATE$ then
        $event \leftarrow$ createEvent($notification$);
        add $event$ to $eventList$;
        addEventToHistory($modelHistory$, $event$,
          $lineNumber$);
        identify ignored lines and fill ignoreList;
        $lineNumber \leftarrow lineNumber + 1$;
    else if $getEventType(notification) = Event.ADD$ then
        ...
    end
end
```

The basic algorithm to capture the change events is showed in Alg. 1. Basically, when and operation is applied to a model, an instance of such notification facilities execute a specific method to notify that there has been a change on the model (an event has just been triggered). We can then filter the notification based on its event type by using *getEventType(notification)* function get the type of the event carried by the *notification* and compared to specific

[4]http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/common/notify/Notification.html
[5]http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/util/EContentAdapter.html

event types that we want to filter (lines 2 and 7). If the specific event types are met, we can then execute further operations.

The algorithm creates an *event* object by gathering certain information from the notification, such as affected object, affected feature (attribute or reference), event type, and operand's value (line 3). These information is useful to fill the model history in subsection 4.2. The algorithm then add the *event* into an *eventList* (line 4). The *eventList* can be further persisted into a CBP representation [8], such as the one in Lst. 2. The algorithm also add the event into a modelHistory using the *addEventToHistory* procedure (line 5) (*addEventToHistory* is discussed in the following subsection). After that, the algorithm tries to identify lines that can be ignored and then add them to the ignore list (line 6) (discussed in section 5). The algorithm also receives *lineNumber* as a variable to track the line number of the events filtered. Since we specify every event is mapped only to one line in CBP, *lineNumber* has to be increased by 1 after every event recording into the history list (line 7). With such filter, we can adjust to collect only events that fits with our purpose as well as their line numbers.

4.2 Model History

Events captured in the Alg. 1 are stored into a history list – a particular data structure and a set of operations that tracks objects, their events, and their occurrence in a CBP representation. We name it a model history (Fig. 3). A *ModelHistory* can have many *ObjectHistory*, which is reflection that a model can have more than one object as its elements.*ObjectHistory* has an attribute object that identify an instance of the *ObjectHistory* belongs to specific object. The attribute *isMoved* is a flag used to identify the state of the object if the object is already affected by a *move* operation (discussed in subsection 5.2). Each *ObjectHistory* consists of one or more *EventHistory* to reflect that an object can be involved in different types of events/operations. The *EventHistory* records a set of *Line* that contains lines where an object and its events occur in a CBP representation. The *Line* has attributes *lineNumber* and *value* that hold its line number in the CBP representation and the value of involved operand respectively.

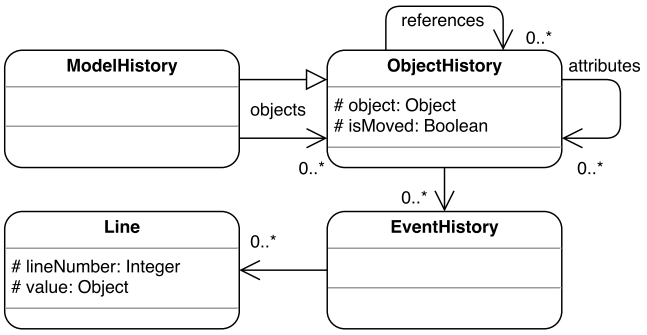

Figure 3: A class diagram of recording object history.

To save events captured in Alg. 1, we follow steps in Alg. 2. The algorithm requires three input: *modelHistory*, *event*, *lineNumber*. Inside the procedure *addEventToHistory*, the algorithm starts by retrieving the *object* affected by the occurring event from the *event*

Algorithm 2: An algorithm to save events captured in Alg. 1 into a model history.

```
input: an object of ModelHistory modelHistory, an object
       of Event event, a value of Integer lineNumber
1  procedure addEventToHistory(modelHistory, event,
    lineNumber)
2  begin
3      object ← getObject(event);
4      if object does not have objectHistory in modelHistory
        then
5          objectHistory ← createObjectHistory(object,
            modelHistory);
6      else
7          objectHistory ← getObjectHistory(object,
            modelHistory);
8      end
9      affectedFeature ← getAffectedFeature(event);
10     if affectedFeature is Attribute then
11         addEventToAttributeHistory(objectHistory, event,
            affectedFeature, lineNumber);
12     else if featureObject is Reference then
13         addEventToReferenceHistory(objectHistory, event,
            affectedFeature, lineNumber);
14     end
15     eventType ← getEventType(event);
16     if eventType does not have eventHistory in
        objectHistory then
17         eventHistory ← createEventHistory(eventType,
            objectHistory);
18     else
19         eventHistory ← getEventHistory(eventType,
            objectHistory);
20     end
21     value ← getValue(event);
22     line ← createLine(lineNumber, value);
23     add line to eventHistory;
24 end
```

using the get *objectFunction* (line 3). The procedure then checks if the *object'sobjectHistory* does not exists yet in the *modelHistory* (line 4). If it is true then it creates a new one (line 5). If it is false then it retrieves the *objectHistory* from the *modelHistory* (line 7).

Lines 9 to 14 are specified to handle events that modify any feature – an attribute or a containment reference – of the object (for example, set a value to an attribute, add an object to a containment reference). From the *event*, we can get the *affectedFeature* using *getAffectedFeature* function. If the *affectedFeature* is attribute then the algorithm executes procedure *addEventToAttributeHistory* (lines 10 and 11). If the *affectedFeature* is reference then the algorithm executes procedure *AddEventToReferenceHistory* (lines 12 and 13). Both *addEventToAttributeHistory* and *AddEventToReferenceHistory* are procedures that works similarly to procedure *addEventToHistory* except that they create and add object history instances, as well

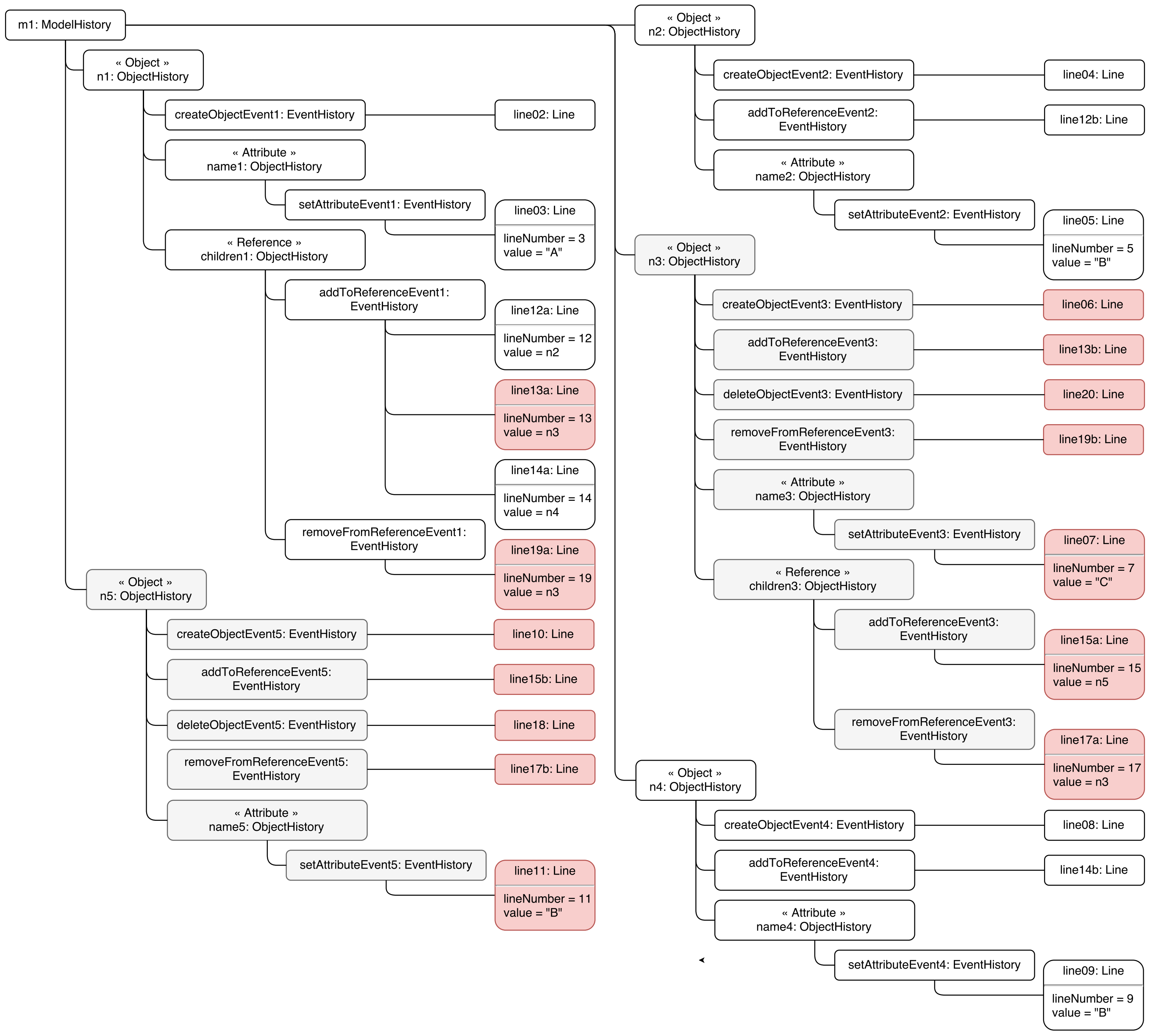
m1: ModelHistory
« Object » n1: ObjectHistory
createObjectEvent1: EventHistory
line02: Line
« Attribute » name1: ObjectHistory
setAttributeEvent1: EventHistory
line03: Line
lineNumber = 3 value = "A"
« Reference » children1: ObjectHistory
addToReferenceEvent1: EventHistory
line12a: Line
lineNumber = 12 value = n2
line13a: Line
lineNumber = 13 value = n3
line14a: Line
lineNumber = 14 value = n4
removeFromReferenceEvent1: EventHistory
line19a: Line
lineNumber = 19 value = n3
« Object » n5: ObjectHistory
createObjectEvent5: EventHistory
line10: Line
addToReferenceEvent5: EventHistory
line15b: Line
deleteObjectEvent5: EventHistory
line18: Line
removeFromReferenceEvent5: EventHistory
line17b: Line
« Attribute » name5: ObjectHistory
setAttributeEvent5: EventHistory
line11: Line
lineNumber = 11 value = "B"
« Object » n2: ObjectHistory
createObjectEvent2: EventHistory
line04: Line
addToReferenceEvent2: EventHistory
line12b: Line
« Attribute » name2: ObjectHistory
setAttributeEvent2: EventHistory
line05: Line
lineNumber = 5 value = "B"
« Object » n3: ObjectHistory
createObjectEvent3: EventHistory
line06: Line
addToReferenceEvent3: EventHistory
line13b: Line
deleteObjectEvent3: EventHistory
line20: Line
removeFromReferenceEvent3: EventHistory
line19b: Line
« Attribute » name3: ObjectHistory
setAttributeEvent3: EventHistory
line07: Line
lineNumber = 7 value = "C"
« Reference » children3: ObjectHistory
addToReferenceEvent3: EventHistory
line15a: Line
lineNumber = 15 value = n5
removeFromReferenceEvent3: EventHistory
line17a: Line
lineNumber = 17 value = n3
« Object » n4: ObjectHistory
createObjectEvent4: EventHistory
line08: Line
addToReferenceEvent4: EventHistory
line14b: Line
« Attribute » name4: ObjectHistory
setAttributeEvent4: EventHistory
line09: Line
lineNumber = 9 value = "B"

Figure 4: An object diagram showing the structure of object history of the model in Fig. 2. The red rectangles contain line numbers that are going to ignored when loading CBP.

as their event histories and lines, for the attribute and reference relationships in Fig. 3. The object history instances represents the history of attributes and references that belongs to an object.

After handling affected features, the algorithm retrieves *eventType* that is held by the *event* using *getEventType* function (line 15). If the *eventType*'s *eventHistory* does not exists in the *objectHistory* then the algorithm creates a new *eventHistory* of the *eventType* in the *objectEvent* using *createEventHistory* function (lines 16 and 17). Otherwise, it retrieves the *eventHistory* from the *objectHistory* (lines 18 and 19). Using *getValue* function, the algorithm gets any operand's *value* used in the event (the *value* can be null if the event does not use any value) (line 21). The algorithm then uses the *lineNumber*, the *value*, and *createLine* function, to create an instance *line* of class Line and adds it into *eventHistory* (lines 22 and 23).

The operations that produce Lst. 2, if they are recorded into ModelHistory, yields structure depicted by the object diagram in Fig. 4. The red rectangles are the Line objects that contain information of line numbers that are going to be ignored when the CBP in Lst. 2 is loaded. Some of the objects are collapsed, not displaying the lineNumber field, to save space. Their line numbers are reflected on their object names.

5 LOADING TIME OPTIMISATION

Loading all events to generate the end model of a CBP representation is not efficient, since we can have multiple operations that are cancelled by other operations – they do not affect the state of the end model. One way to optimise it is by ignoring those operations. Every time an event occurs, we execute algorithms to identify previous events that are cancelled by an event and add them to an ignore list (Alg. 1, line 6). From the *event* object in Alg. 1, we can identify what kind of operations (e.g. create, add, delete) that they represent. With that information, we can perform certain algorithms, utilising the model history in section 4.2, to identify ignorable lines and put them into the ignore list. We discuss the optimisation of *set* and *unset* operation in subsection 5.1, *add*, *remove*, and *move* operations in subsection 5.2, and *create* and *delete* operations in subsection 5.3.

Algorithm 3: CBP loading optimisation algorithm.

```
  input  : a list of Event $eventList$, a list of integer $ignoreList$
1 begin
2     foreach $event$ in $eventList$ do
3         $lineNumber \leftarrow$ getLineNumber($event$);
4         if $lineNumber$ is not in $ignoreList$ then
5             replay($event$);
6         end
7     end
8 end
```

When a CBP representation is loaded, only events that are not contained in the ignore list that are replayed. In Alg. 3, events that are in the *eventList* are iterated one by one (line 2). The *eventList* is loaded from a CBP representation. The algorithm then uses the *getLineNumber* function to get the *lineNumber* of each *event* (line 3). The *lineNumber* then is checked if it does not exist in the *ignoreList* (line 4). If it is true then the *event* is replayed (line 5), otherwise the *event* is ignored.

5.1 Set and Unset Operations

During a model development, an attribute of an object can be assigned many times. We identify that the last value that is assigned to the attribute is all that matters – the last assigned value is the value that is hold by the attribute in the end model. Previous assignments can be ignored since they do not affect the end version of a model. For example in List. 3, the attribute *name* is assigned *"A"*, and then *"B"*, and then nullified (unset), and finally *"C"*. We can execute only the last assignment (line 5) to optimise the execution, ignoring the previous operations, to produce the same end model as if we executes all the operations.

Listing 3: Example of CBP representation of *name* attribute assignments.

```
1   create node of Node
2   set node.name to "A"     //node.name="A"
3   set node.name to "B"     //node.name="B"
4   unset node.name          //node.name=null
5   set node.name to "C"     //node.name="C"
```

To make the optimisastion possible, the line numbers where the attribute *name* affected in Lst. 3 are stored into its respective object history. For example, lines for *set* and *unset* events in their event histories are *setEventHistory.lines = setEventLines = [[2, "A"], [3, "B"], [5, "C"]]* and *unsetEventHistory.lines = unsetEventLines = [[4, null]]* respectively. Since only the last value of set or unset operations that is significant, using the two lists, we can identify that line 5 is the last line of set and unset operations that is significant for the attribute *name*. Therefore, lines 2 to 4 can be put into the ignore list.

Algorithm 4: Algorithm to identify lines that can be ignored for attribute's *set* and *unset* operations

```
1  struct $Line$
2      Integer $lineNumber$; Var $value$;
3  end
   input  : two lists of Line $setEventLines$ and
            $unsetEventLines$, a list of Integer $ignoreList$
   output : a list of Integer $ignoreList$
4  begin
5      $setLastLine \leftarrow$ getLastLine($setEventLines$);
6      $unsetLastLine \leftarrow$ getLastLine($unsetEventLines$);
7      if $setLastLine > unsetLastLine$ then
8          Add every line number in $setEventLines$ into
             $ignoreList$ except the last line;
9          Add all line numbers in $unsetEventLines$ into
             $ignoreList$;
10     else if $setLastLine < unsetLastLine$ then
11         Add all line numbers in $setEventLines$ into
             $ignoreList$;
12         Add all line numbers in in $unsetEventLines$ into
             $ignoreList$;
13     end
14     return $ignoreList$;
15 end
```

The algorithm to execute this argument is shown in Alg. 4. It takes three inputs: two list of *Line*, *setEventLines* and *unsetEventLines*, and a list of Integer, *ignoreList*. The *Line* is a structure that represents an event. It carries the *lineNumber* of the event in its CBP representation as well as the *value* involved. The *setEventLines* and *unsetEventLines* are lists that contains the *Line*s where the event *setAttributeEvent* and *unsetAttributeEvent* appear in a CBP representation and the operand's value included in the event. The algorithm starts by retrieving the last lines of *setEventLines* and *unsetEventLines* using the *getLastLine* function and assigning them to variables *setLastLine* and *unsetLastLine* (lines 5 and 6). If *setLastLine* is larger than *unsetLastLine* then every line number that is in the *setEventLines* are put into the *ignoreList* excluding the last value since the last line still has to be executed to produce the same end model (lines 7-9). If *unsetLastLine* is larger than *setLastLine* then *unset* operation is the last operation that is executed (line 10). Therefore, there is no need to executes all *set* and *unset* operations – all lines can be ignored. Every line numbers in *setEventLines* and *unsetEventLines* are put into the *ignoreList*. The *ignoredList* is then returned for further processes.

Listing 4: Example of CBP representation of *name* reference assignments.

```
1   create n1 of Node
2   create n2 of Node
3   create n3 of Node
4   create n4 of Node
5   set n1.associate to n2    //node.associate=n2
6   set n1.associate to n3    //node.associate=n3
7   unset n1.associate        //node.associate=null
8   set n1.associate to n4    //node.associate=n4
```

The algorithm (Alg. 4) can also be applied to set and unset operations of a containment reference for CBP described in Lst. 4. For nodes *n1*, *n2*, *n3*, and *n4* are created at the beginning of the creation of its model (line 1-4). The *n1*'s single containment reference, *associate*, is then assigned *n2* and *n3* accordingly (lines 5-6), before nullified or unset (line 7). Finally, the *n1.associate* is set to *n4*.

Using the same optimisation to set and unset operations of containment reference *associate*, we can omit lines 5 to 7 to produce the same end model. In in their event histories, the lines for set and unset operations are *setEventLines = [[5, n2], [6, n3], [8, n4]]* and *unsetEventLines = [[7, null]]* respectively. Since only the last value of set or unset operations that is significant, using the two lists, we can identify that line 8 is the last line of set and unset operations that is significant for the containment reference *associate*. Therefore, lines 5 to 7 can be put into the ignore list.

5.2 Add, Remove, and Move Operations

An attribute can also contains many values; we can add remove values from it. For example (List. **??**), *node* object has *values* attribute that can contains many integer values. We add values 11, 12, and 13 subsequently (line 2-4) to the attribute and remove the value 12 at line 5. We show in the commented parts the states of *node.values*.

Listing 5: Example of CBP representation of attribute *values*'s add and remove operations.

```
1   create node of Node
2   add 11 to node.values      //node.values=[11]
3   add 12 to node.values      //node.values=[11,12]
4   add 13 to node.values      //node.values=[11,12,13]
5   remove 12 from node.values //node.values=[11,13]
```

The execution of these operations can be optimised by ignoring the add and remove operations of the value 12 (lines 3 and 5) since we can produce the same end model only by adding 11 and 13 (line 2, 4). When events in List. 5 are executed, they populate line lists for add and remove event histories of attribute *values*, *addEventLines = [[2,11], [3,12], [4,13]]* and *removeEventLines = [[5,12]]*. Using the two lists, we can identify the line number of lines that can be put into the ignore list, producing *ignoreList = [3, 5]*. The algorithm to execute this argument is shown in Alg. 5.

The algorithms takes seven inputs: three line lists – addEventLines, removeEventLines, and moveEventLines – that contain the lines of add, remove, and move events, an ignore list *ignoreList*, the variable *operandValue* to hold the value used in an event, a flag variable *attributeIsMoved*, and an object of Feature *feature* which can be an attribute or containment reference. The algorithm starts by checking the value of *attributeIsMoved* to the member of the

Algorithm 5: Algorithm to identify lines that can be ignored for *add*, *remove*, and *move* operations.

```
struct Line
    Integer lineNumber; Anytype value;
end
input : two lists of Line addEventLines, removeEventLines, and moveEventLines, a list of
        Integer ignoreList, a variable of Anytype operandValue, a variable of Boolean
        attributeIsMoved, an object of Feature feature
output: a list of Integer ignoreList
begin
    if attributeIsMoved = false then
        filteredAddLines ← filterLinesByValue(addEventLines, operandValue);
        filteredRemoveLines ← filterLinesByValue(removeEventLines, operandValue);
        addLastLine ← getLastLine(filteredAddLines);
        removeLastLine ← getLastLine(filteredRemoveLines);
        if addLastLine > removeLastLine then
            Add every line number in filteredAddLines into ignoreList except the last value;
            Add all line numbers in filteredRemoveLines into ignoreList;
        else if addLastLine < removeLastLine then
            Add all line numbers in filteredAddLines into ignoreList;
            Add all line numbers in filteredRemoveLines into ignoreList;
        end
    end
    if feature is empty or feature only has a value then
        if feature is empty then
            Add all line numbers in addEventLines into ignoreList;
        else if feature only has a value then
            Add every line number in addEventLines into ignoreList except the last value;
        end
        Add all line numbers in removeEventLines into ignoreList;
        Add all line numbers in moveEventLines into ignoreList;
        attributeIsMoved ← false;
    end
    return ignoreList;
end
```

attribute's values if they are already moved or not (line). We explain more the use of *attributeIsMoved* later in this section. If the values are not moved yet, the algorithm continues with filtering all the line lists by the operandValue and stored them into filtered line lists: *filteredAddLines* and *filteredRemoveLines*. The algorithm the retrieves the last lines of the filtered line lists using the *getLastLine* function and assigns them to *addLastLine* and *removeLastLine*. If the *addLastLine* is larger than removeLastLine then all line numbers of *filteredAddLines* and *filteredRemoveLines*, except for the last line of *filteredAddLines*, are added into the *ignoreList*. Otherwise, all line numbers in both filtered line lists are added into the*ignoreList*.

The algorithm then proceeds by checking the feature. If the feature is empty then all line numbers in the *addEventLines* are added into the *ignoreList*. If the feature only has one value, all the line numbers also added into the *ignoreList* except the last line. The algorithm then proceeds by adding all line numbers in *removeEventLines* and *moveEventLines* into the *ignoreList*. After that, *attributeIsMoved* is set to false. Finally, the *ignoreList* is returned for further operations.

Listing 6: Example of CBP representation of attribute *values*'s add and remove operations.

```
1  create n1 of Node
2  create n2 of Node
3  create n3 of Node
4  create n4 of Node
5  add n2 to node.children      //n1.children=[n2]
6  add n3 to node.children      //n1.children=[n2,n3]
7  add n4 to node.children      //n1.children=[n2,n3,n4]
8  remove n3 from node.children //n1.children=[n1,n4]
```

The algorithm (Alg. 5) can also be applied to add, remove, and move operations of a containment reference for CBP described in Lst. 6. For nodes *n1*, *n2*, *n3*, and *n4* are created at the beginning of the creation of its model (line 1-4). The *n1*'s multivalue containment reference, *children*, is then added *n2*,*n3*, and *n4* accordingly (lines 5-7), before the removal of node *n3* (line 8).

Using the same optimisation to *add*, *remove*, and *move* operations of containment reference *children*, we can omit lines 6 to 8 to produce the same end model. In in their event histories, the lines for set and unset operations are *addEventLines = [[5, n2], [6, n3], [7, n4]]* and *removeEventLines = [[8, n3]]* respectively. Since the *remove* operation at line 8 makes itself and the *add* operation at line 6 insignificant, lines 6 and 8 can be put into the ignore list.

Listing 7: Example of CBP representation of attribute *values*'s move operations.

```
1  create node of Node
2  add 11 to node.values           //node.values=[11]
3  add 12 to node.values           //node.values=[11,12]
4  add 13 to node.values           //node.values=[11,12,13]
5  move from 0 to 1 in node.values //node.values=[12,11,13]
6  remove 12 from node.values      //node.values=[11,13]
```

Listing 8: Example of optimised CBP representation of attribute *values*'s move operations.

```
1  create node of Node             //(1)
2  add 11 to node.values           //(2) node.values=[11]
3  add 13 to node.values           //(4) node.values=[11,13]
4  move from 0 to 1 in node.values //(5) node.values=[13,11]
```

The '*if attributeIsMoved = false*' condition at line 5 in Alg. 5 indicates that removing previous add and remove events can only be applied if no move operation has been executed previously to the attribute. This condition is required, since after optimisation, some operations are already removed and therefore makes some values may not exist, which changing the indexes of other values. Consequently, replaying all the events may not produce the same end model with the non-optimised CBP representation.

To illustrate this case, we compare the end model loaded from a CBP representation (List. 7) to another model loaded from and optimised CBP representation (List. 8) – the same representation but has been optimised. The optimisation removes line 3 and 6 in List. List. 7 and when loaded back again the end models are not same. The non-optimised CBP representation produces *node.values=[11,13]* while the optimised one produces *node.values=[13,11]*. When *remove 12* is executed (List. 7 line 6), the optimisation ignores the *add 12* operation (List. 7 line 3) and thus makes the index of 13 is 1 in the optimised CBP, while the index of 13 is 2 in the non-optimised CBP, and produces different end model when *move from 0 to 1* (List. 8 line 4) is executed.

The *attributeIsMoved*'s state is set to true when a move operation is applied within it's attribute and the number of of it's attributes' values is more than 1. The *attributeIsMoved*'s state is set to false again when the attribute is empty or only has a value (5, line 18-26) since any move operation does not affect the indexes of the attributes' values.

5.3 Create and Delete Operations

The examples of object's create event has been demonstrated in several Listings in this paper where *create* is the keyword that denotes the event in a CBP representation. The create operation is only executed once per object. It cannot be created more than once. Meanwhile, the example of object's delete event is exhibited in Lst. 2. When an object is deleted, it means that the object is completely removed from the model – no longer exists. Therefore, all events (create, delete, set, unset, add) related to the object can be ignored – there is no need to create the object and all its attributes' events can be ignored as well.

If Lst. 2 is optimised by removing node n3 (n5 is removed first since n5 is contained in n3), then the otimisation produces the CBP representation as in Lst. 9. The optimisation ignores 10 lines (lines 6, 7, 10, 11, 13, 15, 17, 18, 19, and 20) since those lines are related to nodes *n3* and *n5* and ignoring the lines still produces the same end model to the one produced by the non-optimised CBP representation (Lst. 2).

Listing 9: Change-based representation of the model of Figure 2 after removal of node *n5*.

```
1  session s1                  //(1)
2  create n1 of Node           //(2)
3  set name of n1 with "A"     //(3)  n1.name="A"
4  create n2 of Node           //(4)
5  set name of n2 with "B"     //(5)  n2.name="B"
6  create n4 of Node           //(8)
7  set name of n4 with "D"     //(9)  n4.name="D"
8  add n2 to n1.children       //(12) n1.children={n2}
9  add n4 to n1.children       //(14) n1.children={n2,n4}
10 session s2                  //(16)
```

We use Alg. 6 to determine lines that are ignored after a *delete* operation. The algorithm takes two inputs, *deletedObject*, an variable of object that is deleted, and *ignoreList*, a list of Integer that contains ignored line numbers. After some processes, the algorithm will return the *ignoreList* as its output. The algorithm starts by checking whether the *deletedObject* is already moved or not (line 2), if it is not then it is safe to remove all lines that refer to the object (line 3). Otherwise, no action is executed. The algorithm then retrieves all event histories *eventHistoryList* that refer to the object (line 4)and iterates through each event history (line 5-8). For every event history *eventHistory* (line 5), the algorithm retrieves its lines *lineList* (line 6) and put all their line numbers into the *ignoreList* (line 7). After that, the algorithm continues to iterate through all its attributes and put all lines of their events into the *ignoreList* as well (lines 12-15). As a result, the next time the CBP representation (Lst. 2) is loaded, the loading process can refer to the *ignoreList* whether certain lines in the CBP representation are ignored or not.

Algorithm 6: Algorithm to identify lines that are ignored after *delete* operations

```
   input  : a variable ofObject deletedObject, a list of Integer
            ignoreList
   output : a list of Integer ignoreList
1  begin
2      objectIsMoved ← isObjectMoved(deletedObject);
3      if objectIsMoved = false then
4          eventHistoryList ←
             getAllEventHistories(deletedObject);
5          foreach eventHistory in EventHistoryList do
6              lineList ← getLines(eventHistory);
7              Add all line numbers in lineList into ignoreList;
8          end
9          attributeList ← getAllAttributes(deletedObject);
10         foreach attribute in attributeList do
11             eventHistoryList ←
                 getAllEventHistories(attribute);
12             foreach eventHistory in EventHistoryList do
13                 lineList ← getLines(eventHistory);
14                 Add all line numbers in lineList into
                     ignoreList;
15             end
16         end
17     end
18     return ignoreList;
19 end
```

For example, based on Lst. 2 and Fig. 4, at the moment when node *n5* is deleted, it already had 5 instances of *eventHistory*: 4 instances (*createObjectEvent5*, *addToReferenceEvent5*, *deleteObjectEvent5*, and *removeFromReferenceEvent5*) that are contained directly in its object history and 1 instance (*setAttributeEvent5*) is contained in its attribute *name* object history. Those instances has line lists that contain line numbers as follows: *createObjectEvent5.lines = [[10]]*, *addToReferenceEvent5.lines = [[15]]*, *deleteObjectEvent5.lines = [[18]]*, and *removeFromReferenceEvent5.lines = [[17]]*, *setAttributeEvent5.lines = [[11, "B"]]*. Using Alg. 6, deleting node *n5* means filling the *ignoreList* with all the line numbers contained by its event histories. Thus, the deletion produces *ignoreList = [10, 11, 15, 17, 18]*. Using the same approach, the deletion of node n3 adds more line numbers and produces *ignoreList = [6, 7, 10, 11, 13, 15, 17, 18, 19, 20]*.

6 LOADING TIME TEST

We have implemented a prototype[6] of the change-based model persistence upon the Eclipse Modelling Framework and utilise its notification facilities to create a CBP representation. We test the performance of our prototype to gain insight ho much efficiency that can be gained with the optimisation algorithms. We do comparison on loading speed between optimised CBP and non-optimised CBP, and also with XMI, a common standard for model serialisation, as the comparison baseline. The comparison is depicted in Fig. 5.

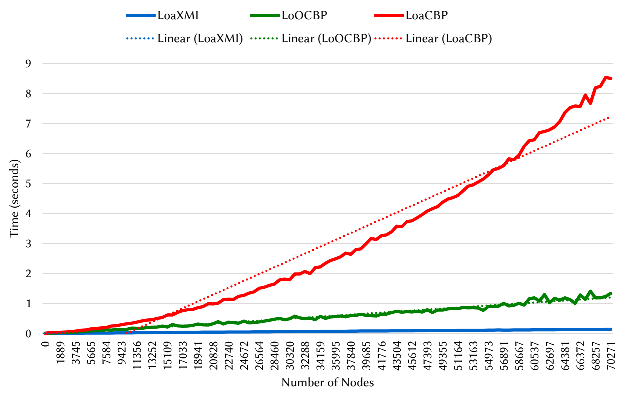

Figure 5: A comparison on loading time between XMI (LoaXMI), optimised CBP (LoOCBP), and non-optimised CBP (LoaCBP).

In performing our comparison, we seek the relationship between number of nodes and time required to all the nodes. The number of nodes increases each time we perform measurement. We perform an iterative measurement that starts with an empty model – no node exists – and then increases the number of nodes by 500 for each succeeding iteration until the number reaches 55,500. In each iteration, first, initial nodes are populated as many as the number of nodes for that iteration.

The model then goes through an alteration process, a process that randomly manipulates the model with a number of different operations. The number of operations is as many as the initial number of nodes. So if the initial number of nodes is 1000 then the number of succeeding operations is 1000. The type of each operation is also randomly selected from a set of possible operation types, which are *set attribute*, *unset attribute*, *add literal value*, *move literal value*, *remove literal value*, *add object value*, *move object value*, *create object*, and d*delete object*. The probability of occurrence of the types are set to ratio 1:1:3:2:1:3:2:1:1 respectively. As an example, based on the ratio, the probability for *delete object* operation to occur is 1/15 for each iteration.

[6]The prototype is available under https://github.com/epsilonlabs/emf-cbp

The population of the initial nodes and the alteration are then serialised as an Epsilon Object Language (EOL) script [4]. This serialisation makes the same produced script to be used across different loading time tests: loading XMI, optimised CBP, and non-optimised CBP.

Including create and delete operations in the succeeding operations makes the number of nodes at the end of each alteration possible to be different from the number of the initial nodes. Thus, we can identify in Fig. 5 that values in x-axis are not exactly the multiple of 500.

Fig. 5 shows that the non-optimised CBP consumes more time than XMI and optimised CBP in loading model. It follows an exponential pattern along the increment of nodes. Our optimisation is proven that optimised CBP is becoming more efficient when the number of loaded nodes is increasing. However, still, it cannot outperform the XMI's loading time, since more time is used to de-serialise the CBP format. Optimising the serialised format is possibly will reduce the loading time. From our simulation, with the specification of probability of random operations mentioned previously, the loading time of optimised CBP is 7.84 (*SD* = 1.47) times slower in average than XMI's loading time. Although it is slower, the loading time for 70,271 nodes is only a bit more than one second which is tolerable for a user when loading a model in a modelling application.

However, models in the real world are most likely different from the random models generated in our loading time testing since real-world models have their own unique characteristics. Thus, the loading time are vary across different models. The optimised CBP may perform better or worse than the one presented in this paper. Moreover, the optimised CBP is not always out perform the non-optimised CBP in every condition. There is condition where optimised CBP is not faster than the non-optimised CBP that is when only *create* operation is performed – no other types of operations, since there is no event that can be ignored.

7 LIMITATIONS

So far, we only address feature – attribute or reference – that can only contains unique members – no duplicate values or objects. Duplicate members means that removal of a value /object does not mean the removal of other same values/objects since values/objects with different positions contained in the feature are perceived as different entity. Therefore, positions of values/objects have to be taken into account as well by the optimisation algorithm. We also only address containment reference – deletion of an object will also delete its contained objects – which is very different from the non-containment one that the deletion of an object does not delete its contained objects. The deletion has to identify non-containment reference so the creation events objects that are referenced should not be ignored. Furthermore, we also have not address features that has default values. Default value of a feature might not trigger any event. Failure to address this might produce different end models.

Moreover, the loading test performed might not reflect the CBP of the construction of real-world models. Each model has its own development characteristics and one feasible way to capture them is to ask modellers to create complex models using different modelling languages, persist their histories, and analyse the results to gain insight whether the optimisation really works for real-world models.

8 CONCLUSIONS

In this paper, we have proposed a change-based, as an alternative to stated-based, persistence as an approach to persist a model. The persistence enables high-performance processing (e.g. transformation, validation) for incremental model by reducing the change identification cost of evolving models. We have illustrated our approach in implementing the persistence as well as its optimisation algorithms to improve its loading time. Based on our performance test on model loading time, optimised change-based persistence outperforms its unoptimised version. However, even though it still cannot outperform the stated-based persistence's loading time, it operates in the range of time that is still tolerable for users.

For future work, persisting models in a change-based format means that model files will keep growing in size during their evolution significantly faster than their state-based counterparts. To address this challenge, (1) we will propose sound change-compression operations (e.g. remove older/unused information) that can be used to reduce the size of a model in a controlled way. (2) We will develop a compact textual format that will minimise the amount of space required to record a change (a textual line-separated format is desirable to maintain compatibility with file-based version control systems). (3) We will propose a hybrid model persistence format which will be able to incorporate both change-based and state-based information.

ACKNOWLEDGMENTS

This work was partly supported through a scholarship managed by XXX.

REFERENCES

[1] Alexander Egyed. 2011. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering* 37, 2 (2011), 188–204.

[2] Frédéric Jouault and Massimo Tisi. 2010. Towards Incremental Execution of ATL Transformations. *ICMT* 10 (2010), 123–137.

[3] Udo Kelter, Jürgen Wehren, and Jörg Niere. 2005. A Generic Difference Algorithm for UML Models. *Software Engineering* 64, 105-116 (2005), 4–9.

[4] Dimitrios S Kolovos, Richard F Paige, and Fiona Polack. 2006. The epsilon object language (EOL). *ECMDA-FA* 6 (2006), 128–142.

[5] Babajide Ogunyomi, Louis M Rose, and Dimitrios S Kolovos. 2015. Property access traces for source incremental model-to-text transformation. In *European Conference on Modelling Foundations and Applications*. Springer, 187–202.

[6] István Ráth, Ábel Hegedüs, and Dániel Varró. 2012. Derived features for EMF by integrating advanced model queries. *Modelling Foundations and Applications* (2012), 102–117.

[7] Bran Selic. 2003. The pragmatics of model-driven development. *IEEE software* 20, 5 (2003), 19–25.

[8] Alfa Yohannis, Fiona Polack, and Dimitris Kolovos. 2017. Turning Models Inside Out. In *FlexMDE@MoDELS (to appear in)*.