
Proceedings Paper:
Yohannis, Alfa, Rodriguez, Horacio Hoyos, Polack, Fiona orcid.org/0000-0001-7954-6433 et al. (1 more author) (2018) Towards efficient loading of change-based models. In: Modelling Foundations and Applications - 14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Proceedings. 14th European Conference on Modelling Foundations and Applications, ECMFA 2018 Held as Part of STAF 2018, 26-28 Jun 2018 Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, FRA, pp. 235-250.

https://doi.org/10.1007/978-3-319-92997-2_15

Towards Efficient Loading of Change-Based Models

Alfa Yohannis, Horacio Hoyos Rodriguez*, Fiona Polack**, and Dimitris Kolovos

Department of Computer Science, University of York, United Kingdom
**School of Computing and Maths, Keele University, United Kingdom
{ary506, dimitris.kolovos}@york.ac.uk
*horacio_hoyos_rodriguez@ieee.org
**f.a.c.polack@keele.a.cuk

Abstract. This paper proposes and evaluates an efficient approach for loading models stored in a change-based format. The work builds on language-independent change-based persistence (CBP) of models conforming to object-oriented metamodelling architectures such as MOF and EMF, an approach which persists a model's editing history rather than its current state. We evaluate the performance of the proposed loading approach and assess its impact on saving change-based models. Our results show that the proposed approach significantly improves loading times compared to the baseline CBP loading approach, and has a negligible impact on saving.

1 Introduction

Conventional approaches for file-based model persistence in metamodelling architectures such as MOF [1] and EMF [2] are state-based – saving the current state of a model. In these approaches, version control and change detection are delegated to external systems. State-based persistence is computationally expensive, as a whole model must be saved and loaded; this can particularly affect large models and collaborative developments.

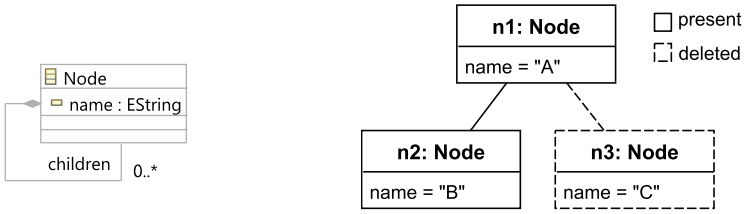
In [3], we proposed change-based persistence (CBP), an approach that persists the full sequence of *changes* made to a model instead of persisting the state. Compared to state-based approaches, CBP supports fast detection of changes, which can speed up model comparison and merging, as well as fast incremental model validation and transformation [4,5]. However, saving the change history of a model results in large, and ever-growing, CBP files. Loading times are also significant, as the loading process has to reconstruct a model's current state from its history [3]. This paper proposes and evaluates an approach that reduces CBP model loading time by avoiding the replaying of historical changes that have no impact on the final state of the model.

The rest of the paper is structured as follows. Section 2 introduces a running example and provides a brief introduction to CBP. Section 3 presents the ap-

proach to speed up model loading and its supporting data structures. Section 4 presents experimental results and evaluation. Section 5 provides an overview of related work, and Section 6 concludes with a discussion on directions of future work.

2 Running Example

To explain model CBP, we use a minimal tree metamodel and an example tree model in Figures 1a and 1b. The metamodel is expressed the Eclipse Modelling Framework (EMF) Ecore metamodelling language, the de-facto standard for object-oriented metamodelling. The example is contrived to avoid unnecessary repetition, whilst providing adequate coverage of the core features of Ecore (classes, single/multi-valued features, references). In this example, a tree model consists of named nodes which can – optionally – contain other nodes (*child* reference).



(a) The tree metamodel (EMF/Ecore).

(b) A tree model that conforms to the metamodel. Node n3 is created and then deleted.

Fig. 1: Running example of a metamodel and a conformant model.

The current state of the model in Figure 1b has two nodes, $n1$, $n2$. The model was constructed by firstly creating the three nodes ($n1$, $n2$ and $n3$) and then nodes $n2$ and $n3$ were then added as children of $n1$. Finally, node $n3$ was deleted.

Listing 1: State-based tree model.

```
1  <Node id="n1" name="A">
2  <children id="n2" name="B"/>
3  </Node>
```

Listing 2: Change-based tree model.

```
1  create n1 of Node
2  set n1.name to "A"
3  create n2 of Node
4  set n2.name to "B"
5  create n3 of Node
6  set n3.name to "C"
7  add n2 to n1.children
8  add n3 to n1.children
9  remove n3 from n1.children
10 delete n3
```

Listings 1 shows the state-based representation of the model, using simplified XMI. Listing 2 shows the change-based representation, using the CBP syntax introduced in [3]. Lines 1-6 of Listing 2 record the creation and naming of the three nodes; lines 7-8 record the addition of $n2$ and $n3$ as children of $n1$; lines 9-10 capture the deletion of $n3$ (the *remove* command removes f $n3$ from its container; the *delete* command completely removes $n3$ from its model). Changes in a CBP representation can be uniquely identified by their line numbers.

The example model history illustrates a case where earlier events (creating *n3* in line 5, naming it in line 6, making it a child of *n1* in line 8, removing it from the container in line 9) are superseded by a subsequent event (deletion of *n3* in line 10). Loading of the current model would arguably be faster if the events in lines 5, 6, 8, 9 and 10 could be ignored.

3 Towards Efficient Loading of Change-Based Models

The flowchart in Figure 2 provides an overview of the editing lifecycle of a CBP model [3], with the proposed extensions shown as starred blocks. A model is loaded (1), edited (2) and saved (3). During editing, the changes made to the model are recorded in a memory-based data structure, serialised and with the latest events appended at the end (4). The change events are persisted into a CBP file every time the model is saved (5). When a model is re-loaded, the current model state is recreated by replaying the events stored in the CBP file (6).

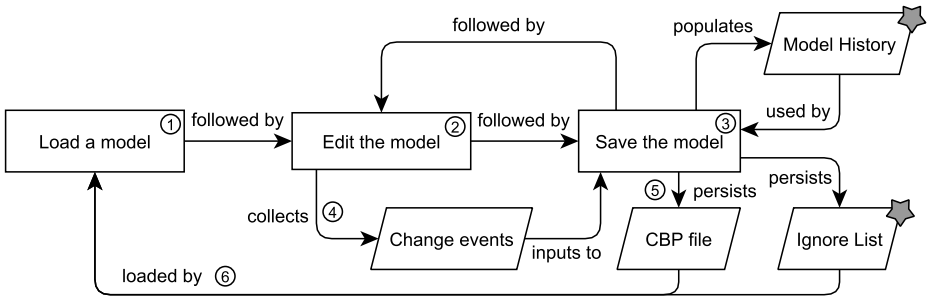


Fig. 2: CBP workflow, with optimised loading elements indicated by starred blocks.

A key principle of CBP is that the editing history is immutable, as this is essential for supporting incremental model management operations. As such, superseded events cannot be simply removed from the CBP file. Therefore, the proposed approach adds two artefacts: a in-memory $ModelHistory$ data structure which aggregates change events per model element, and an $IgnoreList$ file, which persists the position (i.e. line numbers) of superseded events so that the events can be ignored the next time the model is loaded. The Ignore List is saved alongside the CBP file. The rest of this section presents how the Model History is used to detect superseded events and generate the Ignore List.

3.1 Model History

The Model History data structure stores events and their line numbers in a CBP representation. The data can be used to reason about the events of a particular element and to determine which events are superseded. We refer to the line

number in the CBP representation as the *event number*. The proposed data structure is defined in Figure 3 using a class diagram.

A *ModelHistory* has a *URI* attribute to identify the model for which it records changes. A *ModelHistory* can link to many *ElementHistory* objects, each identified by its *element* field which is queried from the model. An *ElementHistory* can link to many *FeatureHistories*, representing the editing histories of individual features – either references or attributes of the element. A *FeatureHistory* has a *type* (attribute or reference) and a *name*, identifying the feature.

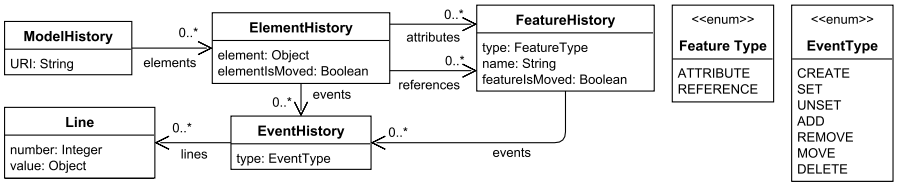


Fig. 3: The class model defining Model History.

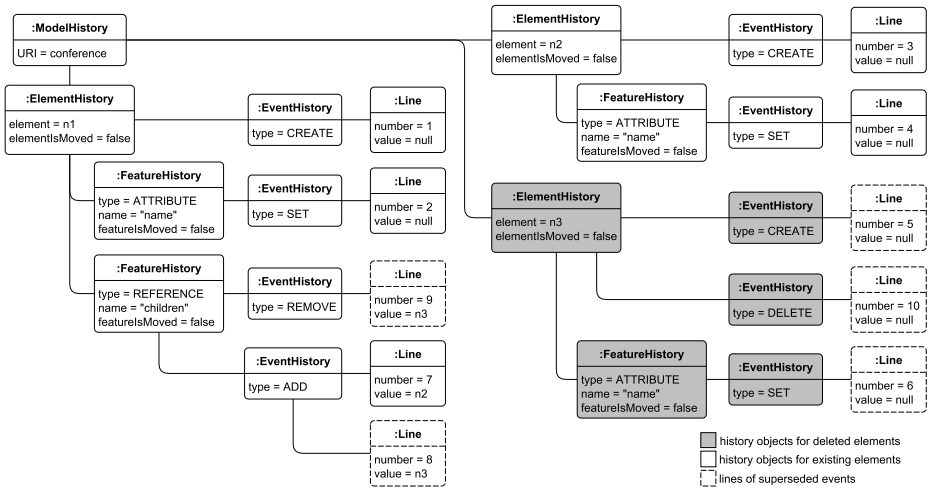


Fig. 4: The object diagram of the CBP model history in Listing 2.

An *EventHistory* represents series of events of the same type; it has an attribute *type* to identify the events' type and can have many *Line*s. A *Line* has a *number* attribute, to record the event number and a *value* that records the element involved in the event (Value is only used for events with types ADD, $REMOVE$ and $MOVE$). Each *FeatureHistory* can have many *EventHistories*, to represent the events that modify the values of the features. Each *ElementHistory* can have many *EventHistories* to represent events that affect the state of the elements (life-cycle and relations to multivalued features). Figure 4 shows an object diagram corresponding to the model in Figure 3 that captures the model history

shown in Listing 2. The grey rectangles are *History* objects related to the deleted node *n3*. The rectangles with the dashed outline are *Line* objects that represent superseded changes.

Next, we present the different strategies used to identify superseded events that will be added to the Ignore List.

3.2 Set and Unset Events

During the lifecycle of a model, a single-valued feature can have its value set (assigned) or unset many times. Each event is persisted, but only the last assigned value needs to be considered. For example, in Listing 3, the feature *name* is set to the value "A", unset, and finally set to the value "B". In the final state of the model, *n1.name* = "B". Thus, only line 4 is significant for the model's final state and therefore lines 2 and 3 can be ignored when loading the model. For a *set* event, all preceding *set* and *unset* events can be ignored, but for an *unset* event, all *set* and *unset* events can be ignored. Executing it does not have any effect on the final state of a model if all the preceding events also have been ignored.

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

```
1  create n1 of Node
2  set n1.name to "A"
3  unset n1.name
4  set n1.name to "B"
```

Listing 4: A CBP representation of attribute *name* assignments.

```
1  create n1 of Node
2  set n1.name to "A"
3  set n1.name to "B"
4  unset n1.name
```

Based on the Listing 3, our approach creates an instance of $ElementHistory$ $n1$ which contains an instance of $FeatureHistory$ *name*. The $FeatureHistory$ *name* consists of two $EventHistory$ instances, with types SET and $UNSET$ (the instances are named *set* and *unset* respectively for brevity). The *set* records the *Line* instances that hold the event numbers where the *set* events, and similarly for *unset*.

From Listing 3, we can thus infer that $name.set.lines = \{2, 4\}$ and $name.unset.lines = \{3\}$. The event numbers in both lists are used to determine that the events represented by lines 2 and 3 are superseded by that in line 4, which is a *set* event, giving an $ignoreList = \{2, 3\}$. By the same process, for Listing 4, we can reason that $name.set.lines = \{2,3\}$ and $name.unset.lines = \{4\}$. However, this case, the highest-numbered event is an *unset*, all so line numbers are put into the *ignoreList* ($ignoreList = \{2, 3, 4\}$) (*unset* event can be ignored along with all preceding *set* and *unset* events).

3.3 Add, Remove, and Move Events

For a multi-valued feature, add, remove, and move events can be called many times, to modify the feature. If an element is added to the feature, moved mul-

tiple times, and finally removed, then all the element's preceding events can be ignored, as long as the order of the feature's elements is not changed.

Listing 5 shows an example without a *move* event. In the Listing, nodes $n1$, $n2$, and $n3$ are added to the *children* feature of p (lines 5-7), In the latest state of the model, *children* only contains $n1$ and $n3$. As a result, the loading process could ignore the events that represent the *add* and *remove* events on $n1$.

Listing 5: A CBP of add and remove operations.

```
1  create p of Node
2  create n1 of Node
3  create n2 of Node
4  create n3 of Node
5  add n1 to p.children
6  add n2 to p.children
7  add n3 to p.children
8  remove n2 from p.
       children
```

Listing 6: A CBP representation of add, move, and remove operations.

```
1  create p of Node          //children=[]
2  create n1 of Node         //children=[]
3  create n2 of Node         //children=[]
4  create n3 of Node         //children=[]
5  add n1 to p.children      //children=[n1]
6  add n2 to p.children      //children=[n1,n2]
7  add n3 to p.children      //children=[n1,n2,n3]
8  move 0 to 1 in p.children //children=[n2,n1,n3]
9  remove n2 from p.children //children=[n1,n3]
```

To create the Ignore List for the Listing 5, we can deduce that *children.add.lines* = {{5, $n1$}, {6, $n2$} {7, $n3$}} (5 is the line number and $n1$ is the value) and *children.remove.lines* = {{8, $n1$}}. Since $n2$ is removed from its containing feature (line 8), then executing its preceding add and remove events is unnecessary. Note that we retain the *create* event (line 3) as $n2$ has not been deleted from the model – only removed from its containing feature. We can iterate through the add and move structures to identify the events on $n2$ that should be removed, resulting in the *ignoreList* = {6, 8}.

Listing 6 shows an example with a *move* event[1]. A *move* event is inserted at line 8 thus makes the *remove* event of $n2$ moves to line 9. With the introduction of this *move* event, we now have the *children.add.lines* = {{5, $n1$}, {6, $n2$} {7, $n3$}}, *children.move.lines* = {{8, $n1$}}, and *children.remove.lines* = {{9, $n2$}}. In the final state of the model, the *children* should have the $n1$ and $n3$ in order, *children* = [n1, n3].

However, executing the previous strategy naively leads to an erroneous final state. Using *ignoreList* = {6, 8} produced by the naive strategy leads to a different order of $n1$ and $n3$ in the final state of the model where *children* = [n3, n1] as shown by the naive optimised CBP in Listing 7. To overcome this problem, $*IsMoved$ flags in Figure 3 is used to sign features and elements if they have been moved – the flags are set to *true*. If an element's $*IsMoved$ flag is true then all of its line numbers related to *add*, *move*, *remove* events cannot be put into the *ignoreList*. The flags are set to $false$ if the feature is empty.

Listing 7: A naive optimised CBP representation of original CBP representation in Listing 6 .

```
1  create p of Node          // children = []
2  create n1 of Node         // children = []
3  create n2 of Node         // children = []
```

[1] The commented parts show the end states of *children* after each event

```
4  create n3 of Node           // children = []
5  add n1 to p.children        // children = [n1]
6  add n3 to p.children        // children = [n1, n3]
7  move 0 to 1 in p.children   // children = [n3, n1]
```

3.4 Create and Delete Events

When an element is deleted, it is completely removed from the model. Therefore, all previous events (*create*, *set*, *unset*, *move*, *add*, *remove*, *delete*) on features of element can be ignored, along with all events on the element's features. For example, when node *n3* in Listing 2 is deleted, the events in lines 5-6 and 8-10 are superseded. If the Listing 2 is optimised – some of its events are ignored – when loading, it runs as if the Listing 8 are executed.

Listing 8: Change-based representation of the model in Figure 1b after removal of node $n3$.

```
1  create n1 of Node
2  set n1.name to "A"
3  create n2 of Node
4  set n2.name to "B"
5  add n2 to n1.children
```

Using the Listing 2, we can construct the structure of histories that are related to element $n3$ as follows: $n3.create.lines = \{5\}$, $n3.name.set.lines = \{6\}$, $n1.children.add.lines = \{\{7, n2\}, \{8, n3\}\}$, $n1.children.remove.lines = \{\{9, n3\}\}$, and $n3.delete.lines = \{10\}$. Thus, when element $n3$ is deleted, by iterating through all these history structures, all line numbers associated with $n3$ can be identified and added to *ignoreList* producing $ignoreList = \{5\ 6, 8, 9, 10\}$ so they can be ignored in the next model loading.

4 Performance Evaluation

We developed the proposed efficient loading approach on top of the original CBP implementation[2] from [3] and evaluated our approach's model loading performance, as well as its memory footprint and its impact on the time required to save changes made to CBP models. The evaluation was performed on Intel® Core™ i7-6500U CPU@2.50GHz 2.59GHz, 12GB RAM, and the Java™ SE Runtime Environment (build 1.8.0_162-b12).

Given that CBP is a very recent contribution and we are not aware of any existing datasets containing real-world models expressed in a change-based format, we have used synthetic change-based models for the evaluation of our experiments.

[2] The prototype, tests, and data used in the evaluation are available under https://github.com/epsilonlabs/emf-cbp and https://goo.gl/1zUBQC for reproducibility

The synthetic models were derived from real-world cases: the BPMN2 [6,7] and Epsilon [8,9] software projects, and the United States article [10] on Wikipedia (the article is further referred as Wikipedia). For the first two projects, for each version of the cases, we used MoDisco [11] to generate a UML2 [12] model that reflects its source code. For the Wikipedia article, a model that conforms to the Modisco XML metamodel [13] was generated. Since these cases have many versions – represented by commits/revisions, different models of the versions can be generated, and to some degree, they reflect the time-ordered changes of the cases. The synthetic change-based model for each case was derived by comparing an initially-empty running model to different versions of the case's models sequentially. All identified differences were then reconciled by performing a unidirectional merging to the running model. All changes made to the running model during the merging process were captured and persisted into a CBP file. EMF Compare was used [14] to perform the comparison and merging.

Using the synthetic models, we performed performance evaluation on loading time, saving time, and memory footprint for both loading and saving. To compare the loading time, we ran the optimised and original (baseline) CBP algorithms to reconstruct the current state of each of the three models (the results are shown in Figure 5). As discussed in Section 3, optimised CBP also does extra work when saving the changes to a model, in order to save time (relative to original CBP) when loading a model. To analyse the performance effect of optimisation activities, we, therefore, compared the overall time required to save a new version of the models described above, after one single change has been made (The results are shown in Figure 6). We also compare the memory footprints for both loading and saving since the optimised CBP approach also requires the maintenance of an additional in-memory data structure that keeps track of element and feature editing histories (see Figure 7 and 8 for the results).

For each combination of dimensions (loading time, saving time, loading memory footprint, saving memory footprint), persistence types (original CBP, optimised CBP, and XMI), and cases (BPMN2, Epsilon, and Wikipedia), we performed measurement 22 times. The results of the measurement enabled us to perform the Welch's t-test [15] to find the significance of the comparisons for each case. We used a significance level of 5%. If t-test' $p\text{-}value < 0.05$, we rejected the null hypothesis – the *means* of the compared persistence types are equal (H_0) – and accepted the alternative hypothesis – the *means* of the compared persistence types are not equal (H_1).

For loading and saving time, we measured the delta time required to complete the loading and saving. For memory footprint, we measured the delta of memory used before and after loading and saving completes. The results are presented below.

4.1 Data Description

Table 1 summarises events, elements and saved versions for the Epsilon, BPMN2, and Wikipedia cases. *Total Events* is the numbers of events that were produced

Table 1: Description of change-based models generated for evaluation.

Model	Total Events	Ignored Events	Elements	Total Versions	Processed Versions
BPMN2	1.2 million	1.1 million	62,062	192	192 (100.0%)
Epsilon	2.6 million	1.8 million	79,459	3,037	727 (23.9%)
Wikipedia	11.5 million	7.8 million	12,144	37,996	3,100 (8.2%)

by our approach in generating a change-based model for each case. *Ignored Events* is the number of superseded events that do not need to be replayed when reloading the models. *Elements* is the number of elements contained in each model. *Total Versions* is the number of commits/revisions made to the cases, taken from the git repositories or Wikipedia at the time this evaluation performed. *Processed Versions* is the number of commits/revisions that were processed to produce change-based models: since the comparison between versions takes considerable time, not all versions are processed here.

4.2 Model Loading Time

This subsection presents the results of the loading time measurement of change-based models for each pair of the persistence types and cases, and the t-test results of their comparisons (Table 2 and Figure 5).

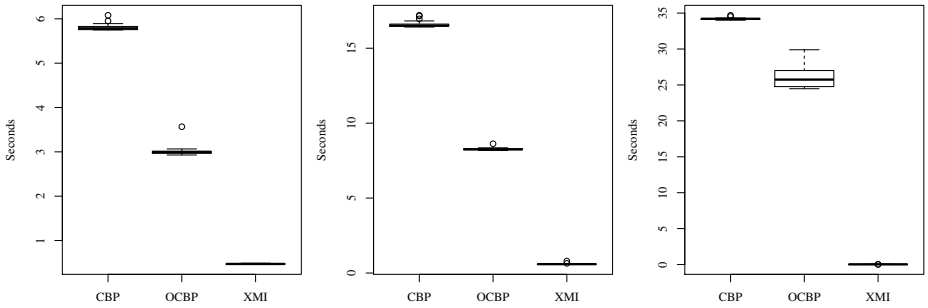
Table 2: The t-test results of loading time comparison between original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Load Time (s)			**BPMN2 Load Time**			
CBP	5.81	0.08	CBP vs. XMI	315.95	21.46	< 0.05
OCBP	3.02	0.13	CBP vs. OCBP	87.67	35.10	< 0.05
XMI	0.47	0.47	OCBP vs. XMI	93.86	21.18	< 0.05
Epsilon Load Time (s)			**Epsilon Load Time**			
CBP	16.60	0.23	CBP vs. XMI	324.18	22.78	< 0.05
OCBP	8.28	0.09	CBP vs. OCBP	160.06	27.48	< 0.05
XMI	31.54	0.05	OCBP vs. XMI	354.52	42.06	< 0.05
Wiki Load Time (s)			**Wikipedia Load Time**			
CBP	34.23	0.145	CBP vs. XMI	1,110.10	21.00	< 0.05
OCBP	26.14	1.583	CBP vs. OCBP	23.90	21.35	< 0.05
XMI	0.02	0.001	OCBP vs. XMI	77.37	21.00	< 0.05

$Mean$ = average, SD = standard deviation, t = t-test's *t-value*, df = degree of freedom, *p-value* = significance, s = the unit is seconds

These loading times show a considerable time saving for optimised CBP: BPMN2 was 48.02% faster, Epsilon 50.12% faster, and the Wikipedia page 23.63% faster than in the original CPB implementation (all optimised CBP's *means* are smaller than all original CBP's *means*), which has a positive correlation to the number of ignored events. All the t-test results also show that loading times for all the persistence types are significantly different (all the *p-values* < 0.05).

For reference, we also compare CBP loading with the execution time for loading the equivalent state-based model in XMI. Figure 5 shows that, even with the improvements delivered by the new algorithm, loading change-based models is still significantly slower than loading a state-based model (all XMI's means are smaller than other persistence types' means).



(a) BPMN2 (b) Epsilon (c) Wikipedia

Fig. 5: Results for loading a model in original CBP (CBP), optimised CBP (OCBP), and for loading a state-based (XMI) representation.

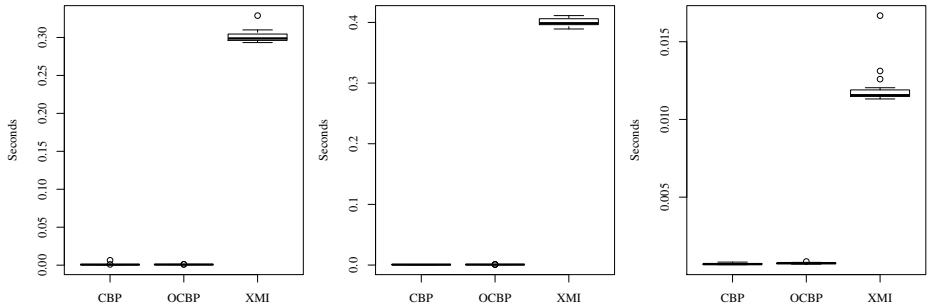
4.3 Model Saving Time

This subsection presents the results of the saving time measurement of change-based models for each pair of the persistence types and cases, and the t-test results of their comparisons (Table 3 and Figure 6). As discussed in [3], CBP loading time penalties are balanced against the benefits that CBP brings, in terms of persisting changes (saving time).

Table 3: The t-test results of saving time comparison between original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Save Time (*s*)			**BPMN2 Save Time**			
CBP	0.00097	123e-5	CBP vs. XMI	-175.58	22.01	< 0.05
OCBP	0.00081	12e-5	CBP vs. OCBP	0.62	21.38	0.54
XMI	0.30122	793e-5	OCBP vs. XMI	-177.76	21.01	< 0.05
Epsilon Save Time (*s*)			**Epsilon Save Time**			
CBP	0.00069	3.4e-5	CBP vs. XMI	-6.01	21.00	< 0.05
OCBP	0.00080	8.0e-5	CBP vs. OCBP	160.06	28.24	< 0.05
XMI	0.40025	595e-5	OCBP vs. XMI	-314.80	21.01	< 0.05
Wiki Save Time (*s*)			**Wikipedia Save Time**			
CBP	0.00071	4.9e-5	CBP vs. XMI	-46.19	21.08	< 0.05
OCBP	0.00075	4.1e-5	CBP vs. OCBP	-3.48	40.77	< 0.05
XMI	0.01195	114e-5	OCBP vs. XMI	-46.01	21.06	< 0.05

Mean = average, *SD* = standard deviation, *t* = t-test's *t-value*, *df* = degree of freedom, *p-value* = significance, *s* = the unit is seconds



(a) BPMN2 (b) Epsilon (c) Wikipedia

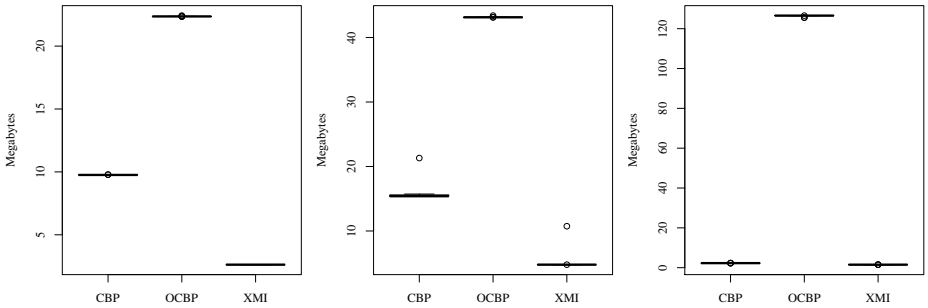
Fig. 6: A comparison on time required for persisting an event between original CBP (CBP), optimised CBP (OCBP), and XMI.

As shown in Table 3 and Figure 6, the performance of the two CBP implementations is not very different. Since the significance level is 5%, only the BPMN2 case that fails. However, the difference between the *means* of its original CBP (0.97 ms) and optimised CBP (0.81 ms) is small. This indicates that the cost of the extra work in the optimised CBP algorithm is negligible. On the other hand, both CBP implementations are significantly faster at saving changes than state-based XMI (the *means* of both CBP implementations are smaller than XMI's *means*, and both CBP implementations have $p\text{-}values < 0.05$ when compared to XMI). This is expected, as the CBP implementations only need append the last changes to the existing model file (their performance is thus relative to the number of changes since the last save), while the XMI implementation needs to reconstruct an XML document for the entire state of the model, and replaces the contents of the model file every time (and hence its performance is relative to the size of the entire model).

4.4 Memory Footprint

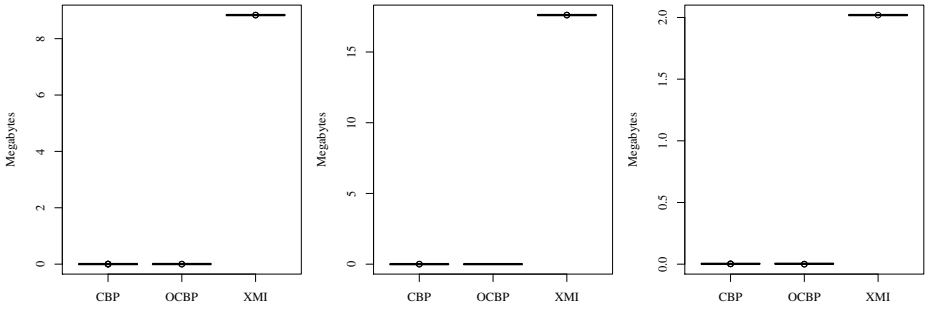
Here we present the results of measuring the memory footprint after loading models (Table 4 and Figure 7) and persisting single changes (Table 5 and Figure 8) using the models from the three cases. The results show the significant memory overhead of the extra data structure when loading models (all the *means* of optimised CBP are greater than all the *means* of original CBP and all comparisons between both CBPs show $p\text{-}values < 0.05$, Table 4). Both CBPs are also outperformed by XMI in terms of memory footprint when loading models (all the *means* of XMI are smaller than all the *means* of both CBPs and all comparisons against XMIs show all $p\text{-}values < 0.05$, Table 4). In loading, XMI uses significantly less memory than the optimised CBP representation and performs slightly better than the original CBP.

In terms of saving, both CBP implementations persist a single change faster than XMI indicated by their *means* that are smaller than the *means* of XMI,



(a) BPMN2 (b) Epsilon (c) Wikipedia

Fig. 7: A comparison on memory footprint after loading a model between original CBP (CBP), optimised CBP (OCBP), and XMI.



(a) BPMN2 (b) Epsilon (c) Wikipedia

Fig. 8: A comparison on memory footprint after persisting an event between CBP, optimised CBP, and XMI.

and all the CBPs' t-tests with XMI show that their differences are significant at $p\text{-}value < 0.05$ (Table 5). The optimised CBP has a larger memory footprint than the original CBP since the means of the optimised CBP for all cases are greater than the means of the original CBP. However, their memory footprints are not very different. Even though the BPMN2 and Epsilon cases have $p\text{-}values < 0.05$, the differences of the *means* of their original and optimised CBPs are small, and the Wikipedia case also shows $p\text{-}value > 0.05$ on its original CBP vs. optimised CBP comparison.

4.5 Threats to Validity and Limitations

In this work, we have only tested the algorithms on synthesised models which may not be representative of the complexity and interconnectedness of models in other domains. Diverse characteristics of models in different domains can affect the effectiveness of the algorithm and therefore yield different outcomes. So far, CBP optimisation only supports ordered and unique features. Support for duplicate values means that removal of an item does not necessarily result in

Table 4: The t-test results of memory footprint comparison after loading a model between original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Load Memory (M)			**BPMN2 Load Memory**			
CBP	9.76	76.0e-4	CBP vs. XMI	4,392.5	21.22	< 0.05
OCBP	22.36	0.015	CBP vs. OCBP	-3,695.7	32.28	< 0.05
XMI	2.63	5.5e-4	OCBP vs. XMI	6,572.4	21.06	< 0.05
Epsilon Load Memory (M)			**Epsilon Load Memory**			
CBP	15.74	1.248	CBP vs. XMI	28.16	41.99	< 0.05
OCBP	43.15	0.056	CBP vs. OCBP	-102.9	21.08	< 0.05
XMI	5.05	1.271	OCBP vs. XMI	140.49	21.08	< 0.05
Wiki Load Memory (M)			**Wikipedia Load Memory**			
CBP	2.29	2.4e-4	CBP vs. XMI	4,523.5	25.16	< 0.05
OCBP	126.48	0.29	CBP vs. OCBP	-2,009.3	21.00	< 0.05
XMI	1.52	7.6e-4	OCBP vs. XMI	2,021.8	21.00	< 0.05

$Mean$ = average, SD = standard deviation, t = t-test's t-$value$, df = degree of freedom, p-$value$ = significance, M = the unit is megabytes

the item not being present in the feature value. Additional information must be captured to persist the number of copies and positions of the feature members to properly generate the ignore list.

4.6 Discussion

For the original CBP loading, the total time required to load a model is $T_{CBP} = T_E + T_O$, where T_E is the total time required to complete executing all events, and T_O is the total time needed to complete other required routines (e.g. initialisation, reading files). For the optimised CBP loading, the total time to load a change-based model is reduced by the total time saved-up by ignoring superseded events T_I, that is $T_{OCBP} = T_E + T_O - T_I$. Thus, it is expected that optimised CBP can load a model faster than original CBP. This statement is in accordance with our finding in Section 4.2 that the total saved-up loading time corresponds to the number of ignored events. However, it still requires more investigation to determine the degree of their correlation, which will be addressed in our future work.

5 Related Work

There are several non-XMI approaches to state-based model persistence, using relational or NoSQL databases. For example, EMF Teneo [16] persists EMF models in relational databases, while Morsa [17] and NeoEMF [18] persist models in document and graph databases, respectively. None of these approaches provides built-in support for versioning and models are eventually stored in binary files/folders which are known to be a poor fit for text-oriented version control systems like Git and SVN. Connected Data Objects (CDO) [19], provides support for database-backed model persistence as well as collaboration facilities,

Table 5: The t-test results of memory footprint comparison after saving an event between original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Save Memory (M)			**BPMN2 Save Memory**			
CBP	0.0023	6.3e-5	CBP vs. XMI	-489,170	41.49	< 0.05
OCBP	0.0029	80e-5	CBP vs. OCBP	-3.22	21.26	< 0.05
XMI	8.84	5.6e-5	OCBP vs. XMI	-51,180	21.21	< 0.05
Epsilon Save Memory (M)			**Epsilon Save Memory**			
CBP	0.0025	18.8e-6	CBP vs. XMI	-4.3e+6	21.00	< 0.05
OCBP	0.0031	279.9e-6	CBP vs. OCBP	-10.131	21.19	< 0.05
XMI	17.61	2.4e-6	OCBP vs. XMI	-295,090	21.00	< 0.05
Wiki Save Memory (M)			**Wikipedia Save Memory**			
CBP	0.0025	1.9e-5	CBP vs. XMI	-391,970	40.52	< 0.05
OCBP	0.0028	84.1e-5	CBP vs. OCBP	-1.75	21.02	0.094
XMI	2.0194	1.5e-5	OCBP vs. XMI	-11,245	21.01	< 0.05

$Mean$ = average, SD = standard deviation, t = t-test's t-$value$, df = degree of freedom, p-$value$ = significance, M = the unit is megabytes

but its adoption necessitates the use of a separate version control system in the software development process (e.g. a Git repository for code and a CDO repository for models), which introduces fragmentation and administration challenges [20]. Similar challenges arise in relation to other model-specific version control systems such as EMFStore [21].

6 Conclusions and Future Work

This paper proposes an efficient algorithm and supporting data structures for loading change-based models. Performance is evaluated on synthesised models, with comparison against the existing change-based implementation, and state-based XMI. Our results show considerable savings in terms of loading time with a negligible impact on saving time, but at the cost of a higher memory footprint. In future, we intend to evaluate CBP against state-based persistence on real complex models. We also plan to investigate the impact of change-based model persistence on the performance of change detection, model merging, and conflict resolution in the context of collaborative modelling. Meanwhile, the CBP approach can be further optimised to consume less memory and speed up parsing, such as using binary format instead of text. We are also exploring a hybrid persistence representation that offers a combination of state-based and change-based persistence.

Acknowledgements. This work was partly supported by through a scholarship managed by *Lembaga Pengelola Dana Pendidikan Indonesia* (Indonesia Endowment Fund for Education).

References

1. OMG: Metaobject facility. http://www.omg.org/mof Accessed: 2018-02-21.
2. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Eclipse Series. Pearson Education (2008)
3. Yohannis, A., Polack, F., Kolovos, D.: Turning models inside out. In: Proceedings of the 3rd Workshop on Flexible Model Driven Engineering co-located with ACMIEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2017). (2017)
4. Ráth, I., Hegedüs, Á., Varró, D.: Derived features for EMF by integrating advanced model queries. In: Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings. (2012) 102–117
5. Ogunyomi, B., Rose, L.M., Kolovos, D.S.: Property access traces for source incremental model-to-text transformation. In: Modelling Foundations and Applications - 11th European Conference, ECMFA 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-24, 2015. Proceedings. (2015) 187–202
6. Eclipse: Mdt/bpmn2. http://wiki.eclipse.org/MDT/BPMN2 Accessed: 2018-01-15.
7. Eclipse: Bpmn2 git. https://git.eclipse.org/c/bpmn2/org.eclipse.bpmn2.git/ Accessed: 2018-02-19.
8. Eclipse: Epsilon. https://www.eclipse.org/epsilon/ Accessed: 2018-02-12.
9. Eclipse: Epsilon git. https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git Accessed: 2018-02-19.
10. wikiedia: United states. https://en.wikipedia.org/wiki/United_States Accessed: 2018-02-19.
11. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: A model driven reverse engineering framework. Information & Software Technology **56**(8) (2014) 1012–1032
12. Eclipse: Mdt/uml2. http://wiki.eclipse.org/MDT/UML2 Accessed: 2018-01-15.
13. Eclipse: Xml metamodel. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.xml.doc%2Fmediawiki%2Fxml_metamodel%2Fuser.html Accessed: 2018-02-19.
14. Eclipse: Emf compare. https://www.eclipse.org/emf/compare/ Accessed: 2018-01-15.
15. Welch, B.L.: The generalization of 'student's' problem when several different population variances are involved. Biometrika **34**(1/2) (1947) 28–35
16. Eclipse: Teneo. http://wiki.eclipse.org/Teneo Accessed: 2017-10-15.
17. Espinazo-Pagán, J., Cuadrado, J.S., Molina, J.G.: Morsa: A scalable approach for persisting and accessing large models. (2011) 77–92
18. Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: Neoemf: a multi-database model persistence framework for very large models. In: Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016. (2016) 1–7
19. Eclipse: CDO the model repository. https://www.eclipse.org/cdo/ Accessed: 2017-10-15.

20. Barmpis, K., Kolovos, D.S.: Evaluation of contemporary graph databases for efficient persistence of large-scale models. Journal of Object Technology **13**(3) (2014) 3: 1–26
21. Koegel, M., Helming, J.: Emfstore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. (2010) 307–308