myjournal manuscript No.
(will be inserted by the editor)

Model Differencing and Conflict Detection using Change-based Persistence

Alfa Yohannis[1,3], Horacio Hoyos Rodriguez[1], Fiona Polack[2], Dimitris Kolovos[1]

[1] Department of Computer Science, University of York, United Kingdom
[2] School of Computing and Maths, Keele University, United Kingdom
[3] Department of Computer Science, Kalbis Institute, Indonesia

Received: 1 July 2019 / Revised version: 1 August 2019

Abstract Comparison – difference identification and conflict detection – of large models can be time-consuming since every element has to be visited, matched, and compared with its respective element in other models. This can result in bottlenecks in collaborative modelling environments, where identifying differences between two versions of a model is desirable. Reducing the comparison process to only the elements that have been modified since a previous known state (e.g. previous version) could significantly reduce the time required for large model comparison. This paper presents how change-based persistence can be used to localise the comparison of models so that only elements affected by recent changes are compared and to substantially reduce comparison and differencing time (up to 90% in some experiments) compared to state-based model comparison.

1 Introduction

In modelling and model management, it is common to find that many versions or variants of a model exist. These versions are commonly persisted as snapshots of the model at a given point in time, in a state-based format such as XMI. Model comparison activities can be applied to the different versions of a model to highlight their differences: changes in properties values, new elements, etc. However, comparing versions of large file-based[1] models in a state-based format can be computationally expensive since both versions of the model need to be loaded in memory in their entirety before their elements can be matched and diffed.

In our previous work [1–3], we proposed change-based persistence (CBP) as an alternative approach to state-based persistence of EMF models [4]. Instead of persisting models as XMI snapshots, in the proposed approach models are persisted as a complete history of changes. We demonstrated the substantial performance benefits of CBP in terms of saving changes to large models [1] as well as a method for reducing model loading time compared to naively replaying all recorded change events [3] to reconstruct the state of a change-based model. In this paper, we demonstrate how a change-based representation also enables much more efficient and performant model comparison between versions of the same model. Our experiments, presented in Section 9, demonstrate savings of the order of 90% for (relatively) small changes made to large models.

This paper is structured as follows. Section 2 provides an overview of our previous work on change-based model persistence. Section **??** discusses state-based model comparison. Section 4 presents our change-based approach to speed up model comparison and its implementation. Section 9 reports on the results of evaluation experiments used to evaluate the proposed approach. Section 11 provides an overview of related work, and Section 12 concludes with a discussion on directions for future work.

2 Change-based Persistence

CBP is an alternative approach to state-based persistence (SBP) of models. Instead of persisting snapshots of the state of a model – which is the default behaviour of frameworks such as EMF – CBP persists the entire history of change events of a model [2]. For example, in SBP approach, when we save the UML class diagram in Fig. 1a in standard XMI format, we only record the last state of the model, as shown in List. 1. In contrast, when we develop the same model in CBP approach, all the change events generated from modifying the model are captured

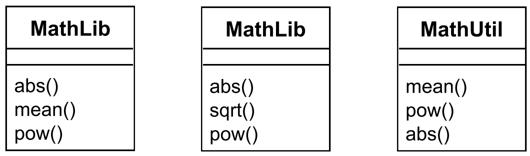
Send offprint requests to:

[1] Persisting models in databases involves its own challenges which have been discussed extensively in the literature. For the rest of the paper, we are only concerned with file-based models and we return to database-backed model representations in Section 11.

and persisted in the model file as shown in List. 2[2]. Each change event contains information about the type of the operation applied as well the as values, elements, or features involved. Replaying the change events in List. 2 produces the same eventual model as in Fig. 1a.

Listing 1: The simplified XMI of the model in Fig. 1a.

```
<uml:Class id="x" name="Math">
<operation id="a" name="abs"/>
<operation id="b" name="mean"/>
<operation id="c" name="pow"/>
</uml:Class>
```



(a) origin (b) left (c) right

Figure 1: Different versions of a model.

Listing 2: The pseudo-formatted CBP of the model in Fig. 1a.

```
create x type Class
set x.name to "Math"
create a type Operation
set a.name to "abs"
create b type Operation
set b.name to "mean"
create c type Operation
set c.name to "pow"
add a to x.operations at 0
add b to x.operations at 1
add c to x.operations at 2
```

3 State-based Model Differencing

In a collaborative modelling setting, a model can have different versions. Consider the case where an initial version of a model exists in a Version Control System (VCS) server (Fig. 2). Two modellers, Bob and Alice, check out the original model (steps 1 and 2) to their local machines and modify it (steps 3 and 4). Alice then commits her work (original + Alice's changes) to the VCS. Since there is no newer commit on the VCS, the commit process is straightforward (step 5). Bob then decides to also commit his work (original + Bob's changes) to the VCS. However, he needs to merge his work with the current updated version at the VCS since his last checkout. His machine downloads the latest version from the server (step 6), i.e. Alice's version. To merge his and Alice's changes, Bob needs to perform model comparison to check their differences, resolve possible conflicts between the models, and then merge them (step 7). After that, he can push it back to the VCS server.

[2] In our implementation, the change-based format is XML-based.

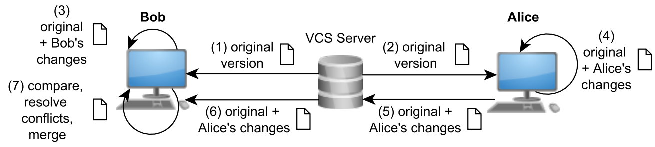


Figure 2: A usecase of CBP in a collaborative modelling.

In a SBP setting, Bob produces the model in Fig. 1b (the left model), and Alice the model in Fig. 1c (the right model) producing XMI files as shown in List. 3 and List. 4 respectively. Before Bob can merge, he must compare the right model with the left model. In state-based comparison, comparing models commonly consists of two steps: *matching* and *diffing*. The matching process establishes matches between the elements of both models, to determine the elements in the left model that correspond to elements in the right model. Generally, the matching process iterates through all the elements of the models being compared and matches them by their identifiers or through a similarity mechanism [5,6].

The diffing process identifies differences between the matched elements [5,6]. Differences between the matched elements and all their features is usually done using a Longest Common Subsequence (LCS) algorithm, e.g. [7].

Listing 3: The simplified XMI of the left model in Fig. 1b.

```
<uml:Class id="x" name="MathLib">
<operation id="a" name="abs/>
<operation id="d" name="sqrt"/>
<operation id="c" name="pow"/>
</uml:Class>
```

Listing 4: The simplified XMI of the right model in Fig. 1c.

```
<uml:Class id="x" name="MathUtil">
<operation id="b" name="mean"/>
<operation id="c" name="pow"/>
<operation id="a" name="abs"/>
</uml:Class>
```

In our example, the matching process in state-based comparison – as performed by EMF Compare [6] – iterates through all the elements of both models and matches them using their identifiers. The matching process yields 3 matches: m_1 = (x, x), m_2 = (a, a), and m_3 = (c, c), and 2 unmatched elements, um_1 = (d, -) and um_2 = (-, b). The diffing process then iterates through all the matches and unmatched elements and uses an LCS algorithm to identify their differences. In the first match, it identifies that the elements x are different in their name and operations features.

The left x's name is "MathLib" while the other x's name is "MathUtil" (diff ds_1). The operations features are different in their contents – the left operations feature does

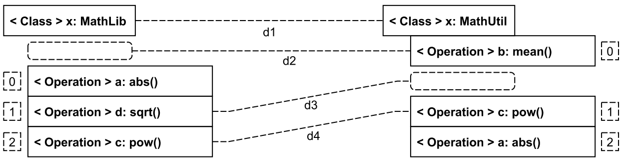


Figure 3: A model comparison of the left and right models in Listings 3 and 4.

not contain element b (diff ds_2), the left operations feature contains element d that does not exist in the right operations (diff ds_3), and the indexes of element c are different in both features (diff ds_4).

Differences are commonly expressed as a list of changes that must be applied to a target model so that it is made equal to a reference model. This paper treats the left model as a reference model and the right model as the target model. This means that differences are expressed as changes applied to the right model so that it equals the left model. To express differences, we use the following terms: LeftContainer, RightContainer, LeftFeature, RightFeature LeftIndex, RightIndex, LeftValue, RightValue, and Kind. The *Container, *Feature, and *Value are the target element, feature, and value involved in a difference (* symbol can be replaced with Left and Right). *Index is the index of a value in a feature. Kind is the type of difference. It can be one of these types: CHANGE, ADD, DELETE, and MOVE. CHANGE means a pair of single-valued features have different values. ADD indicates that a value does not exist in the right model thus it requires the addition of the value. DELETE is the opposite of ADD. MOVE indicates that matched elements differ in terms of their containers, containing features, or indexes. A Container is an element that contains a value. A containing feature is a feature owned by a container in which a value is contained. An index is the position of a value in a containing feature.

Based on these definitions, we can express the result of the diffing process as a tuple:

$$\begin{aligned} ds_n = [&LeftContainer_n, RightContainer_n, \\ &LeftFeature_n, RightFeature_n, \\ &LeftIndex_n, RightIndex_n, \\ &LeftValue_n, RightValue_n, Kind_n] \end{aligned} \tag{1}$$

Thus, $ds_1 =$ [x, x, name, name, 0, 0, “MathLib”, “Mathutil”, CHANGE], $ds_2 =$ [x, x, operations, operations, null, 0, null, b, DELETE], $ds_3 =$ [x, x, operations, operations, 1, null, d, null, ADD], and $ds_4 =$ [x, x, operations, operations, 2, 1, c, c, MOVE]. We can use these information to represent the differences visually as depicted in Fig. 3. Applying these differences as changes to the right model will transform it into the left model.

4 Change-based Approach for Comparing Models

Now lets consider the same example in a CBP setting. The changes made by Bob and Alice are appended to the their local original CBP producing two different CBP representations as displayed in Listings 5 and 6[3] – capturing different courses of modification made by the two modellers. Then the example is the same with Alice committing her changes and Bob wanting to merge Alice's work with his.

Listing 5: The appended changes made by Bob to produce the model in Fig. 1b (left version).

```
12 set x.name from "Math" to "MathLib"
13 create d type Operation
14 set d.name to "sqrt"
15 add d to x.operations at 1
16 remove b in x.operations at 2
17 delete b
```

Listing 6: The appended changes made by Alice to produce the model in Fig. 1c (right version).

```
12 move a in x.operations from 0 to 2
13 set x.name from "Math" to "MathUtil"
```

In CBP, comparison has three phases: event loading, element tree construction, and diff computation. Further, comparison is not performed over all the elements of the model; instead, we only need to compare the last set of changes from the source and reference model. The last set of changes can be easily identified by finding their last common change. A simplified class diagram of our approach's implementation[4] is depicted in Fig. 4. Next, we describe the three phases in detail.

4.1 Event Loading

In the event loading phase, our implementation loads change events recorded in two CBP files into memory. The most important aspect of this phase is the partial loading as only lines starting from the position where the two files are different are loaded. Thus, not the whole model needs to be traversed and loaded. In this case, lines 1-11 in List. 2 are skipped.

Only lines starting from line 12 in Listings 5 and 6 are loaded, yielding two partial – left and right – change-event models.

4.2 Element Tree

An element tree is a representation of the changes of model elements in the source and reference models. It

[3] Both CBPs only present the changes after the last line of the original version (start from line 12).

[4] The source can be found at https://github.com/epsilonlabs/emf-cbp.

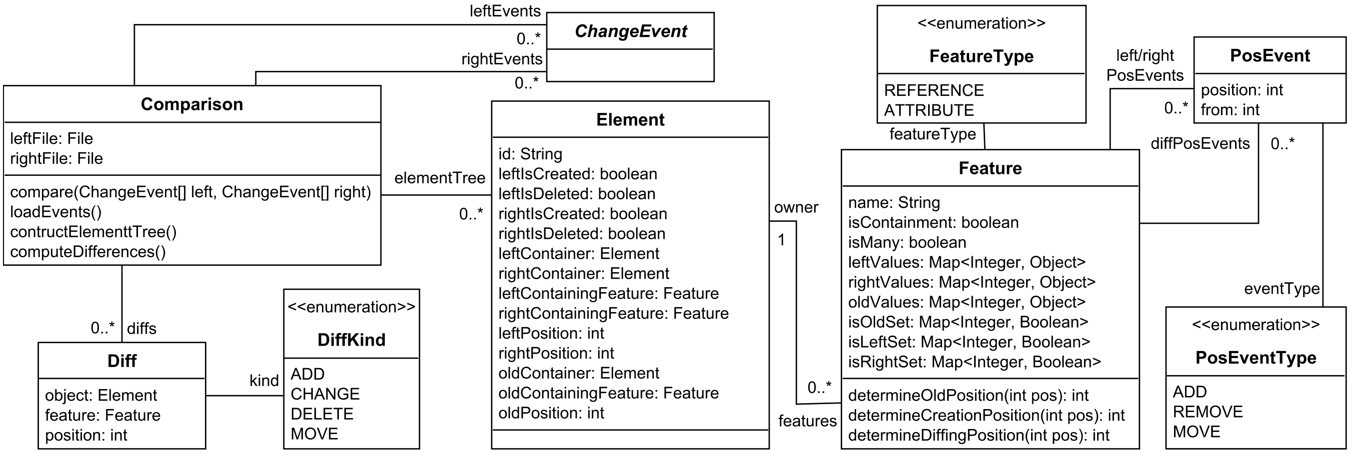


Figure 4: A class diagram showing the core components of the change-based approach to speed up model comparison.

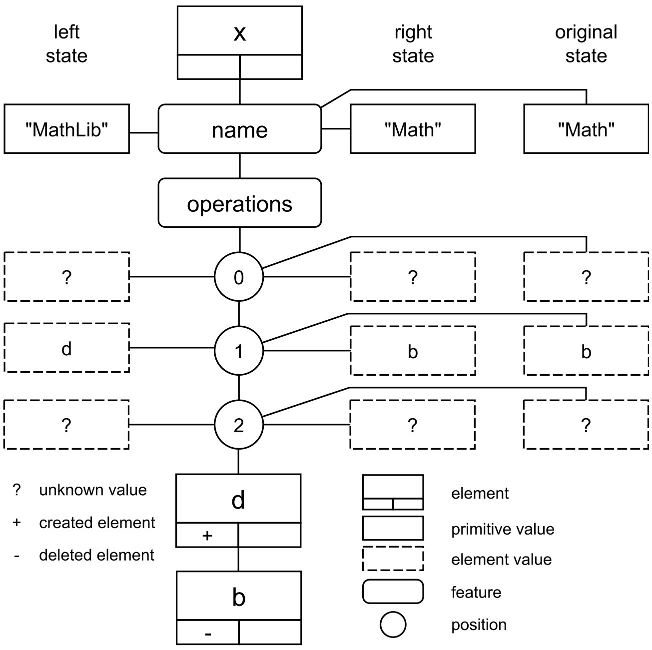


Figure 5: The elementTree after processing all left change events.

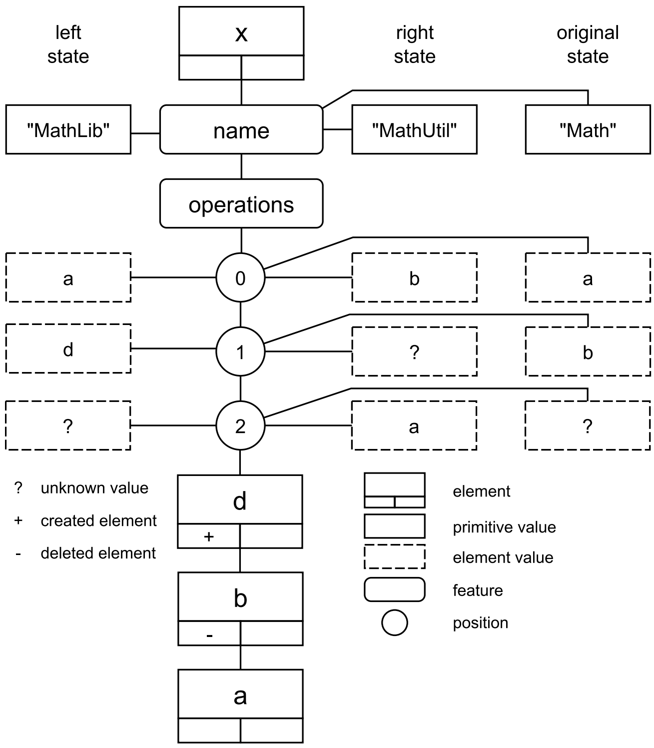


Figure 6: The elementTree after processing all left and right change events.

contains detailed information about elements and their properties.

It contains similar information to that captured in change lists in SBP, but also provides more information about the changes. For example, the element tree can keep track of a feature's old value and element/value's indexes inside multi-valued properties. The element tree only contains the partial states of affected elements of the original, left, and right models as depicted in Figures 5 and 6.

To better understand the construction of an element tree from change events, we use the following running example using both change events in the Listings 5 and 6. We start from the left change events.

4.2.1 Left Side From the first event at line 12, [**set** `x.name` **from** `"Math"` **to** `"MathLib"`], we can identify that an element with id x has existed from the original model. It has a feature name with a value "Math" in the original model that has been changed to "MathLib" in the left model. Since the element x does not already exist in the elementTree, we create its instance of Element and also its feature name. We set the value of the feature name to "MathLib" and also set it to "Math" in the partial state of the original model – it has not been set before. As this feature on the right side also has not been set, we set it to "Math" as well.

At line 13, in the event [**create** d **type** Operation], we can identify that an element with id d has been created. We also update the elementTree to include this element and set the element's flag leftIsCreated to true. In the event [**set** d.name **to** "sqrt"] at line 14, we can identify that element d's feature name has been set to "sqrt". Thus, we update d's feature name in the elementTree. From the event [**add** d **to** x.operations **at** 1] at line 15, we can deduce that element d is added to index 1 in the element x's feature operations. Thus, we assign d to element x's feature operations at index 1 in the elementTree. As d is a new element that only exists in the left model, we do not update changes of this element to the original and right models.

From the event [**remove** b **in** x.operations **at** 2] at line 16, we can identify that there is element b in the original model, but it is deleted in the left model. The index of element b in the original model can be calculated back through the previous change events that have been applied to its feature. Since the previous event is adding element d to index 1 and the index of b is at 2 at the time it is removed, we can deduce that before element d is added, the index of element b is at 1 and is shifted to 2 because of the addition of element d. Therefore, we can conclude that the original index of element b is at 1. Thus, we update the original state of the elementTree by adding element b into the element x's feature operations at index 1.

We perform the same procedure to also add element b to the right state of the elementTree. However, since no change event has been applied to the right side of element x's feature operations, the calculation of element b's index should return the same value as in the original state (line 13, Alg. 1), and thus element b has the same index as in the original state. It is important to notice, in this step, the flag isRightSet (class Feature, Fig. 4) is not set to true since we want the value to be able to be overridden during processing of the right change events. The last event [**delete** b], removes the element b from the left model. Hence, we set the flag leftIsDeleted of element a to true.

Fig. 5 illustrates the state of the elementTree after all left change events have been processed. As can be seen, the elementTree exhibits the partial states of the original, left, and right models at once.

4.2.2 Right Side From the first event at line 12, [**move** a **in** x.operations **from** 0 **to** 2], we can infer that in the right model there is an element with id a positioned at index 2 in the element x's feature operations. Thus, element a – an instance of class Element in 4 – is added to the elementTree and positioned at index 2 of the element x's feature operations. Since the event is a move type and

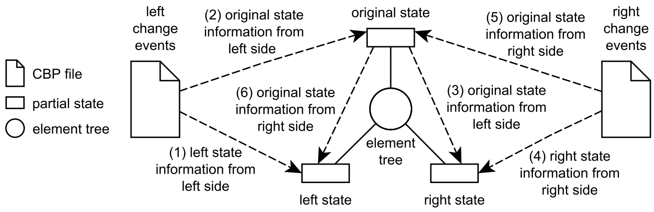


Figure 7: Steps in Element Tree construction.

the new index is larger than its previous index, elements that are between its previous and new indexes are shifted one place down. As element b has already existed in the same feature (the element was added during the process of the left change events) and its index is between element a's movement, the index of element b is shifted down from 1 to 0.

Also since the event's type is move and its previous index is 0 and it is the first event that changes the index of element a, these conditions imply that element a in the original model is positioned at index 0 in the element x's feature operations. Therefore, we add the element a to element x's feature operations in the original state of the elementTree. Since the index 0 in the element x's feature operations has not been set, we also add element a to that index in the right state of the elementTree. From the last event [**set** a.name **from** "Math" **to** "MathUtil"] at line 13, we can infer that in the right model, the value of element a's feature name is "MathUtil". Hence, we set the feature name to "MathUtil" in the right state. We do not apply this operation to the original and left states as they have been set before. Fig. 6 exhibits the state of the elementTree after both sides' change events have been processed.

The construction of the elementTree that we have just explained follows the steps shown in Fig. 7. First, the partial state S_L of the left model in the elementTree is constructed based on the information retrieved from the left change events (step 1). We denote this information as I_{LL}. We can also construct the partial state S_O of the original model using the information related to the original state contained in the left change events I_{OL} (step 2). The information I_{OL} allows us to construct the initial partial state S_R of the right model (step 3). Similarly, using the information from the right change events I_{RR}, we update the partial right state S_R that has been initialised before using the information I_{OL} (step 4), implying that $I_{OL} \cup I_{RR} \to S_R$. Also, information related to the state of the original model from the right change events I_{OR} is used to update the original state (step 5). Thus, we have a partial state of the original model constructed using information from both left and right sides, $I_{OL} \cup I_{OR} \to S_O$. Finally, we also use the information I_{OR} to update the partial state of the left model (step 6), implying that $I_{LL} \cup I_{OR} \to S_L$.

Algorithm 1: Algorithm to construct an element tree from events.

```
   input  : a list of ChangeEvent events
   input  : an enumeration of Side side
   input  : an instance of ElementTree elementTree
   output : an instance of ElementTree elementTree
1  begin
2      foreach event in events do
3          targetElement ←
             getOrCreateNewTargetElement(event,
             elementTree);
4          feature ← getOrCreateNewFeature(event,
             targetElement);
5          value ← getValue(event);
6          previousValue ← getPreviousValue(event);
7          index ← getIndex(event);
8          previousIndex ← getPreviousIndex(event);
9          featureEventList ←
             getFeatureEventList(feature, side);
           // put all values to their proper
              indexes
10         updateTree(targetElement, feature, value,
             index, side);
11         oldIndexes ←
             calculateOldIndex(featureEventList,
             previousIndex, side);
12         if not isCreated(value, side) and not
             isOldValueSet(feature, previousValue,
             previousIndex, side) then
13             setOldValue(feature, previousValue,
                 oldIndex, side);
14             oppFeaEventList ←
                 getOppFeaEventList(feature, side);
15             oppositeIndex ←
                 calculateOppositeIndex(oppFeaEventList,
                 oldIndex, side);
16             if not isDeleted(value, side) and not
                 isOppositeSideValueSet(feature, value,
                 oppositeIndex, side) then
17                 setOppositeSideValue(feature, value,
                     oppositeIndex, side);
18             end
19         end
20         addEventToFeatureEventList(event,
             featureEventList);
21     end
22     return elementTree;
23 end
```

Alg. 1 describes the steps presented in Fig. 7 in an generic fashion. It iterates through all a model's change events and uses the information contained in them to construct the relevant partial state. The selection of side, left or right change events, that is executed first depends on the Side enumeration value – left or right – passed through the parameter side (the second input parameter). In our implementation, we process the left side first by default. The algorithm also receives an input of the change events events that are to be iterated and the element tree elementTree that has been instantiated before, and then returns the elementTree as output after updating it.

For each event in the events, we collect information needed to build up the elementTree (lines 3-9), such as targetElement, feature, value, previousValue, index, and previousIndex. The targetElement is the element modified by a change event (e.g. x and d in List. 5). This targetElement – an instance of class Element in Fig. 4 – is retrieved from the elementTree if it already exists. Otherwise, a new element is created and added to the elementTree (line 3). In this step we also set the flags *IsCreated and *IsDeleted of the element in Fig. 4. For example, if the type of the event is create then *IsCreated is set to true. The feature – an instance of class Feature in Fig. 4 – represents the target element's feature (e.g. name and operations in List. 5) modified by a change event. It is retrieved from the targetElement's feature list, and a new one is created and added to the targetElement's feature list if it has not existed yet (line 5).

The value is the value assigned to the feature in a change event (line 5, Alg. 1). The value can be the type of Element (e.g. elements b and d, lines 17-18, List. 5) or primitive (e.g. the string "MathLib" at line 14 in the List. 5). The previousValue represents the previous value of the modified feature (line 6, Alg. 1). The previousValue is not defined if no previous value has been assigned. For value and previousValue with type Element, the elements that they represent are retrieved from the elementTree, and if they do not exist, new instances are created. If the type is primitive, the value is treated as it is. Not every change event has a value, particularly event with type create or delete which only modify a target element not the element's feature.

The index is the index assigned by a change event to a value in a feature, while previousIndex is the previous index of the value (lines 7-8, Alg. 1). In one change event, we can get both index and previousIndex or only one of them depending on the type of the change event. For example, we can only obtain that the index of d is 1 (line 17 in List. 5) as the change event type is add. In a remove change event, we can only get the previousIndex of b, that is 2 (line 17 in List. 5), as the element does not exist anymore in the left model. We can obtain both of them only in a move change event as an element is moved from a previous index to a new one (line 14 in List. 6). For a single-valued feature, the index and previousIndex are always 0 as the feature can only contain a single value.

At line 9, we retrieve the featureEventList from the feature to be added later with the current event (line 19). The featureEventList is a list – a history – of change events that have been processed that are specific to the feature on the selected side. Using the obtained targetElement, feature, value, and index, the process then updates the state of the elementTree on the selected side (line 10).

After that, it calculates back the original index of a value using the featureEventList and previousIndex (line 11). If the value at oldIndex in the feature has not been set, then the algorithm sets the feature with the previousValue at the oldIndex in the partial state of the original model (lines 12-13). At lines 14-18, the algorithm also does the same thing to the opposite side – if the current side is left then it is right.

4.3 Diff Computation

Using the elementTree presented in Fig. 6, we can determine the difference between the left and right models without having to compare all their elements and features. After the elementTree has been constructed, we iterate through elements and features of the elementTree and use the flags, containers, containing features, and indexes on both sides of each element and value to identify differences between both left and right models. We follow the steps in Alg. 2. The algorithm visits each element and every index of each feature (lines 3-5). At every index, it retrieves the leftValue and rightValue (lines 5-7), passing these, together with the element, feature, and index to a function identifyDiffUsingRules (line 8). The function identifies differences using a set of pre-defined rules which determines differences diffs based on the states of flags of an element, flags and attributes of the element's feature, values of the feature, and indexes of the values. The obtained diffs are then added to the overall list of differences diffList which is output (line 8-9, 13).

Algorithm 2: Algorithm to determine differences.

```
   input  : an instance of ElementTree elementTree
1  begin
2     diffList ← DiffList();
3     foreach element in elementTree do
4        foreach feature in getFeatures(element) do
5           foreach index in getIndexes(feature)
             do
6              leftValue ← getLeftValue(feature,
                index);
7              rightValue ←
                getRightValue(feature, index);
               // rules starts from here
8              diffs ←
                identifyDiffUsingRules(element,
                feature, leftValue, rightValue,
                index);
9              addToDiffList(diffs,diffList);
10          end
11       end
12    end
13    return diffList;
14 end
```

The first rule (Rule 1) in Alg. 3 is to identify changes in single-valued attributes. A feature has to be of type attribute, both side values have to be different, and the element should have not been created or deleted in both models. The second rule (Rule 2) identifies whether an element is in a different location in both models. The element must not have been deleted and must exist from the previous version – the original model. Also, its containers, containing features, or indexes of the element have to be different on both sides.

We illustrate the principles and use of rules by discussing the rules used to identify differences in the running example, which can be found in Alg. 3. The algorithm is the breakdown of the function identifyDiffUsingRules in Alg. 2. As previously stated, it is important to remember that we use the left model as a reference which means the differences are presented as changes that transform the right model to become equal to the left model.

The third rule (Rule 3) identifies the deletion of an element. If an element in the left model is not created but exists in the model, it means that the element has been there from the previous version – the original model. This also means that the element also exists in the right model, unless it has been deleted. Thus, in order to make the right model equals to the left model, the element has to be deleted also in the right model. The fourth rule (Rule 4) identifies the need for an addition of an element. If an element is created in the left model and has not been deleted, it means that the element should be added also to the right model to make both models equal.

In the running example, when the iteration of the elementTree (Fig. 6) returns feature name, the type of the feature is a single-valued attribute and both sides of the feature are different in their values, this means that the condition of the first rule is met. Thus, we can conclude that in order to make the left value of the feature equal to the right value, we must override the value "MathUtil" with "MathLib"; the type of this difference is CHANGE. When the iteration is at index 0 in the element x's feature operations, we have two values: the leftValue is element a, and the rightValue is element b. As a exists on both sides – all *Created and *Deleted flags are false, and it also has a different index, at 0 in the left state and 2 in the right state. This meets the condition of the second rule. Thus, we can conclude that in order to make the index of element a in the right model equals its index in the left model, element a should be moved from index 2 to 0. Thus, the type of this difference is MOVE.

Element b used to exist but has been deleted from the left model (flags leftIsCreated = false, leftIsDeleted = true); it still exists in the right state (flags rightIsCreated = false, rightIsDeleted = false). This condition satisfies the third rule. Therefore, the element b should be deleted from the right model; the type of this difference is DELETE. We can get only one value when the iteration is at index

Algorithm 3: Some rules to determine differences.

```
input  : an Element element, a Feature feature, a variable leftValue, a variable rightValue, an Integer index
output : a List of Diff diffs
diffs ← createDiffList();
// ...
// Rule 1: a rule to determine a change of a single-valued attribute
if getType(feature) is Attribute and isSingleValued(feature) and leftValue <> rightValue and not
  leftIsCreated(element) and not leftIsDeleted(element) and not rightIsCreated(element) and not
  rightIsDeleted(element) then
    diff ← createNewDiff(element, element, feature, feature, index, index, leftValue, rightValue,
      DifferenceType.CHANGE);
    addDiffToDiffList(diff, diffs);
end
// Rule 2: one of rules to determine movement of an element
if getType(feature) is Containment and not leftIsCreated(leftValue) and not leftIsDeleted(leftValue) and not
  rightIsCreated(leftValue) and not rightIsDeleted(leftValue) and (getLeftContainer(leftValue) <>
  getRightContainer(leftValue) or getLeftFeature(leftValue) <> getRightFeature(leftValue) or
  getLeftIndex(leftValue) <> getRightIndex(leftValue)) then
    diff ← createNewDiff(getLeftContainer(leftValue), getRightContainer(leftValue), getLeftFeature(leftValue),
      getRightFeature(leftValue), getLeftIndex(leftValue), getRightIndex(leftValue), leftValue, leftValue,
      DifferenceType.MOVE);
    addDiffToDiffList(diff, diffs);
end
// Rule 3: one of rules to determine deletion of an element
if getType(feature) is Containment and not leftIsCreated(rightValue) and leftIsDeleted(rightValue) and not
  rightIsCreated(rightValue) and not rightIsDeleted(rightValue) then
    createNewDiff(getLeftContainer(rightValue), getRightContainer(rightValue), getLeftFeature(rightValue),
      getRightFeature(rightValue), getLeftIndex(rightValue), getRightIndex(), rightValue, null,
      DifferenceType.DELETE);
    addDiffToDiffList(diff, diffs);
end
// Rule 4: one of rules to determine addition of an element
if getType(feature) is Containment and leftIsCreated(leftValue) and not leftIsDeleted(leftValue) and not
  rightIsCreated(leftValue) and not rightIsDeleted(leftValue) then
    diff ← createNewDiff(getLeftContainer(leftValue), getRightContainer(leftValue), getLeftFeature(leftValue),
      getRightFeature(leftValue), getLeftIndex(leftValue), getRightIndex(leftValue), null, rightValue,
      DifferenceType.ADD);
    addDiffToDiffList(diff, diffs);
end
// ...
return diffs
```

1 in the element x's feature operations; the leftValue is element d, but the rightValue is unidentified. Thus we only process the leftValue. Element d is only created in the left model (flags leftIsCreated = true, leftIsDeleted = false, rightIsCreated = false, rightIsDeleted = false). This meets the condition of the fourth rule. Thus, to make element d also exist in the right state, we must add it into element x's feature operations at index 1. Therefore, the type of this difference is ADD. At index 2, the element a is skipped because it has been processed already.

Similar to the state-based approach in Section 3, we express identified differences as dc_n = [$LeftContainer_n$, $RightContainer_n$, $LeftFeature_n$, $RightFeature_n$, $LeftIndex_n$, $RightIndex_n$, $LeftValue_n$, $RightValue_n$, $Kind_n$]. Thus, dc_1 = [x, x, name, name, 0, 0, "MathLib", "Mathutil", CHANGE], dc_2 = [x, x, operations, operations, ?, 0, ?, b, DELETE], dc_3 = [x, x, operations, operations, 1, ?, d, ?, ADD], and dc_4 = [x, x, operations, operations, 0, 2, a, a, MOVE]. This change-based approach might produce differences that are different from differences that the state-based approach produces. This can be seen between by comparing ds_4 and dc_4 ($ds_4 \neq dc_4$, [x, x, operations, operations, 2, 1, c, c, MOVE] $\neq$ [x, x, operations, operations, 0, 2, a, a, MOVE]). In the state-based approach, element c has a MOVE difference – it has different index (ds_4), while in the change-based approach, this difference is attributed to element a (dc_4). However, in both approaches, if we resolve their differences by performing all-left-to-right merging – making the right model equal to the left model, both approaches produce two models that are equivalent. In this way, we can check the correctness of the identified differences produced by the change-based approach.

5 Another Running Example

In this section, we introduce another running example to explain how to detect conflicts using the element tree. We use this example to construct a new element tree and to explain how conflict detection is performed in EMF Compare, EMF Store, and our change-based conflict detection. Let's say that there is a project to develop a Role Playing Game (RPG). Jane, as the technical leader, set up the initial model. The records of events during setting up the initial is recorded in the CBP in List. 8a. From the List., we know that she created a class Character that contains an operation attack with three parameters: gem, target, and weapon (lines 1-14). She also created four other classes; Troll (lines 15-16), Giant (lines 17-21), Knight (lines 22-26), and Mage (lines 27-28). She then pushed her work to a change-based version control system. If her work is visualised in state-based format, the model looks like in Fig. 8a.

Listing 7: The recorded events to produce the original model in Fig. 8a (original version).

```
1  create character type Class
2  set character.name to "Character"
3  create attack type Operation
4  set attack.name to "attack"
5  add attack to character.operations at 0
6  create gem type Parameter
7  set gem.name to "gem"
8  add gem to attack.parameters at 0
9  create target type Parameter
10 set target.name to "target"
11 add target to attack.parameters at 1
12 create weapon type Parameter
13 set weapon.name to "weapon"
14 add weapon to attack.parameters at 2
15 create troll type Class
16 set troll.name to "Troll"
17 create giant type class
18 set giant.name to "Giant"
19 create cast type Operation
20 set cast.name to "smash"
21 add cast to giant.operations at 0
22 create knight type Class
23 set knight.name to "Knight"
24 create smash type Operation
25 set smash.name to "smash"
26 add smash to knight.operations at 0
27 create mage type Class
28 set mage.name to "Mage"
```

She then assigned this work to Bob and Alice. Both of them checked out this project to their own machine. Alice then started to continue the model. She then moved parameter target to the first place in operation attack's parameters, because she thought it was more intuitive for programmers to think about the target first than the rest parameters (List. 8, line 29). She also moved operations smash from class Knight to class Giant and cast from class Giant to class Mage as thy are more reasonable to belong to their new classes (lines 30-33). Alice also created a generalisation relationship with id rightGen from class Troll to class Character (34-36). Bob also did the same thing except that his generalisation came with id leftGen (List. 9, line 29-31).

Later on, Jane then informed them that she wanted all good characters should be derived from a general, hero-like class, and the enemy should be the Orcs not Trolls. She also instructed that Bob should focus on developing class Knight and Alice on class Mage. In consequence, Alice then changed the name of class Character from "Character" to "Hero" (the id of class Hero is still character) (line 37). Again, Bob did the same thing. He also changed the name of class Character from "Character" to "Hero" (line 32). Instead of creating a new generalisation relationship, both of them preferred to move the generalisation relationships that they had created to their assigned classes. Alice moved generalisation rightGen from class Troll to class Mage (lines 38-39), and Bob move generalisation leftGen from class Troll to class Knight (lines 33-34). Bob also moved parameter target in operation attack to the last index as he thought setting target as the last parameter was intuitive (line 35), and deleted the class Giant, and unfortunately, he deleted class Giant accidentally (lines 36-40). The class diagrams of Bob and Alice's models are visualised in Figures 8b and 8c respectively. Lastly, Alice changed the name of class Troll to "Orc" (line 40) while Bob changed it to "Ogre" (line 41).

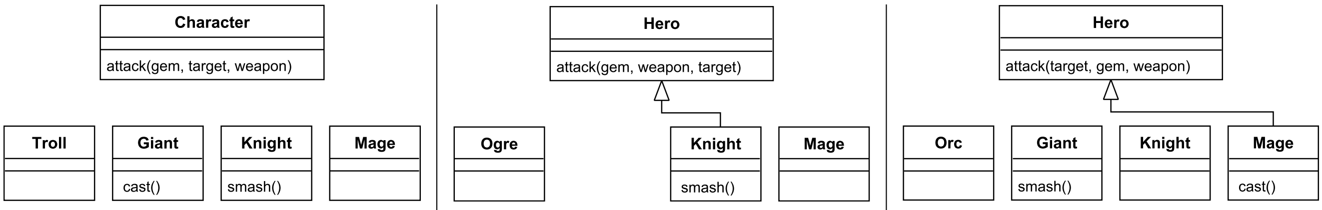
Listing 8: The appended events made by Alice to produce the right model in Fig. 8c (right version).

```
29 move target in attack.parameters from 1 to 0
30 remove smash from knight.operations at 0 composite l1
31 add smash to giant.operations at 0 composite l1
32 remove cast from giant.operations at 1 composite l2
33 add cast to mage.operations at 0 composite l2
34 create rightGen type Generalization
35 set rightGen.general to character
36 set troll.generalization to rightGen
37 set character.name from "Character" to "Hero"
38 remove rightGen from troll.generalization composite
       l3
39 set mage.generalization to rightGen composite l3
40 set troll.name from "Troll" to "Orc"
```

Listing 9: The appended events made by Bob to produce the left model in Fig. 8b (left version).

```
29 create leftGen type Generalization
30 set leftGen.general to character
31 set troll.generalization to leftGen
32 set character.name from "Character" to "Hero"
33 remove leftGen from troll.generalization composite r1
34 set knight.generalization to leftGen composite r1
35 move target in attack.parameters from 1 to 2
36 unset cast.name from "cast" to null composite r2
37 remove cast from giant.operations at 0 composite r2
38 delete cast composite r2
39 unset giant.name from "Giant" to null composite r2
40 delete giant comp r2
41 set troll.name from "Troll" to "Ogre"
```

In Listings 8 and 9, we also introduce composite events – lines with keyword composite – that represent composite operations. Composite operations are operations that should be treated as one composition – identified with



(a) original version (Jane's version) (b) left version (Bob's version) (c) right version (Alice's version)

Figure 8: Three class diagrams of a Role Playing Game.

Table 1: The mapping of elements, features, and values in Fig. 9 to the events that affects them.

Key	Left Events	Right Events
character	ol30, ol32	or35, or37
character.name	ol32	or37
attack	ol35	or29
attack.parameters.target	ol35	or29
target	ol35	or29
troll	ol31, ol33	or36, or38
troll.name	ol41	or40
troll.generalization	ol31, ol33	or36, or38
giant	ol37, ol38, ol39, ol40	or31, or32
giant.name	ol38	
giant.operations.cast	ol37	or32
giant.operations.smash		or31
knight	ol34	or30
knight.generalization	ol34	
knight.operations.smash		or30
mage		or33, or39
mage.generalization		or39
mage.operations.cast		or33
leftGen	ol29, ol30, ol31, ol33, ol34	
leftGen.general	ol30	
rightGen		or34, or35, or36, or38, or39
rightGen.general		or35
smash		or30, or31
cast	ol36, ol37, ol38	or32, or33
cast.name	ol36	

o: operation; l: left side; r: right side; n: line number in the listing

the same composite id. For example, moving an element from a container to another container is composite event since it consists of two operations: removing/unsetting the element from its source container and adding/setting it to its target container (lines 36-37 Listing 8). Using the information contained in CBPs in Listings in CBPs in Listings 8 and 9, we can construct an element tree as depicted in Fig. 9 using the element tree construction method presented in Section 4.2.

During the the construction of the element tree, these events in Listings 8 and 9 are registered to the affected elements, features, and values. This registration forms one-to-many relationships between keys and the events. The keys are `element` for elements, `element-feature` for single-valued features, and `element-feature-value` for multivalued-features. With this mapping, we can trace all events that affects certain elements, features, and values. The mapping of the events in Listings 8 and 9 is in Table 1.

6 Conflict Detection

In state-based model comparison, a conflict detection usually requires two different versions of a model and one shared ancestor – the original version. In change-based model comparison, the original version is not required since it is implicitly contained in the other two versions' change-based persistence. A conflict occurs when an element is identified changed in both versions in reference to its original version. A conflict also occurs when a change from a version is applied first causes another change from the other version cannot be applied due to dependency violation.

Let's say that we have three versions of a model: m_o is the original version, m_l is the left version, and m_r is the right version, where m_o, m_l, $m_r \in M$. Every model consists of elements. Thus, E_O, E_L , and E_R are sets of elements of models M_O, M_L, and E_R respectively, where $E_O = \{e_{o1}, e_{o2}, ..., e_{oa}\}$, $E_L = \{e_{l1}, e_{l2}, ..., e_{lb}\}$, and $E_R = \{e_{r1}, e_{r2}, ..., e_{rc}\}$. Since we detect conflicts up to the level of values in features, an e can also refer to a feature or feature's value instead of only referring to an element. We also have two sets of operations O_L and O_R that changes model m_m to models m_l and m_r respectively, where $O_L = \{o_{l1}, o_{l2}, ..., o_{ld}\}$ and $O_R = \{o_{r1}, o_{r2}, ..., o_{rf}\}$.

6.1 State-based Conflict Detection: EMF Compare

In state-based model comparison, the two sets of operations O_L and O_R are not readily available. They have to be derived trough model differencing. Both sets can be obtained by differencing M_L to M_O and M_R to M_O using an LCS algorithm as explained in Section 3. These two differencing processes produce two sets of differences, D_{OL} and D_{OR}, where $D_{OL} = \{d_{ol1}, d_{ol2}, ..., d_{olg}\}$, $D_{OR} = \{d_{or1}, d_{or2}, ..., d_{orh}\}$, and each difference d is expressed as in (1). These differences can be treated as operations since if we apply these differences as operations to a target model, they transform it to become equivalent to a reference model. For example, applying a set of differences D_{OL} to a model M_O will transform it to model M_L. Therefore, we can say that these differences are equivalent to operations. Thus, we can have two sets of operations, O_L and O_R, which $O_L \equiv D_{OL}$ and $O_R \equiv D_{OR}$. These

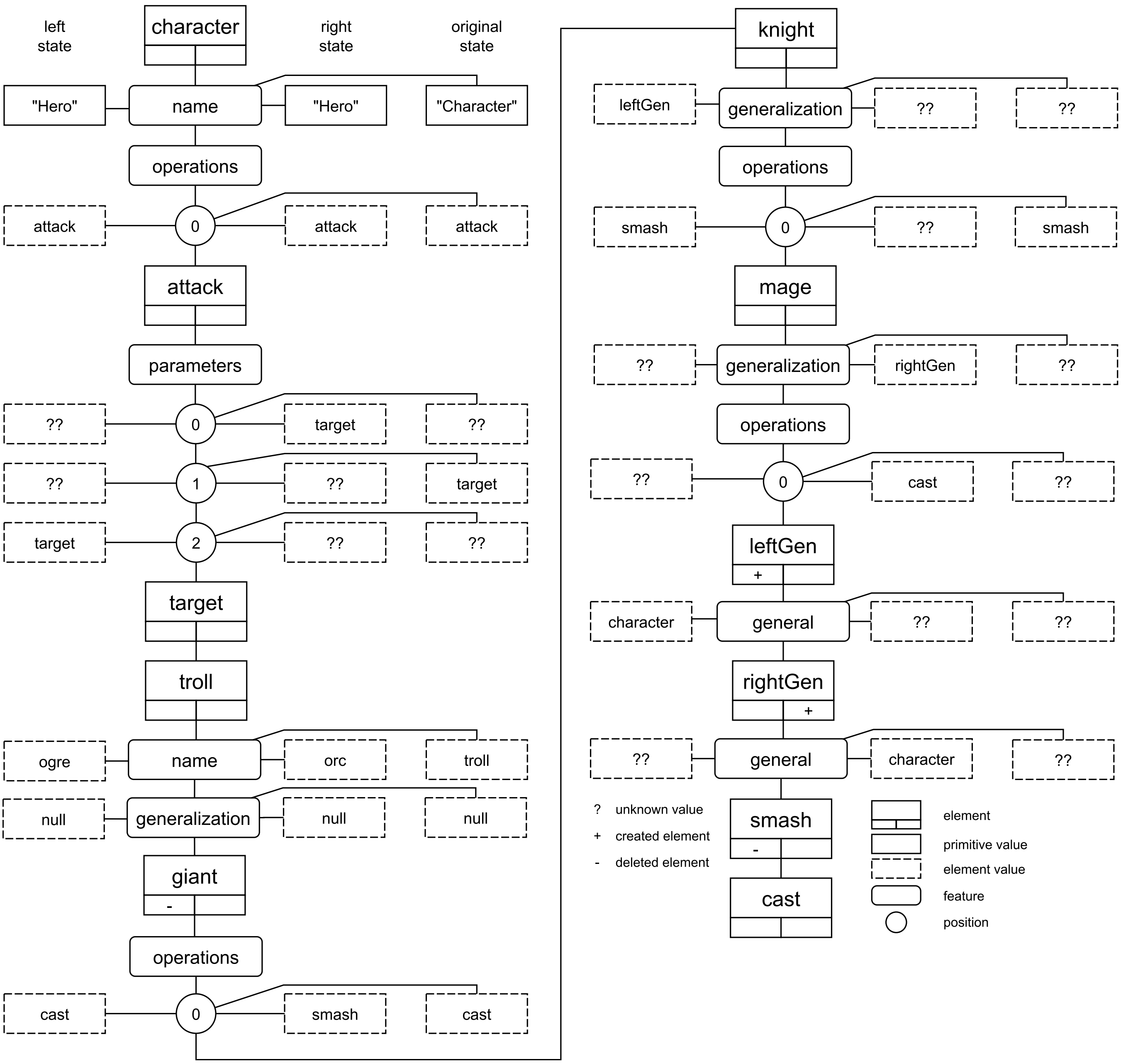


Figure 9: An element tree constructed using information contained in CBPs in Listings 8 and 9.

operations can then be used to detect conflicts using (2), (3), and (4).

If we use EMF Compare to derive O_L from Bob's and Jane's versions in Fig. 8 and present it in the format of change events, we obtain the following list.

Listing 10: The derived change events (operations) made by Bob to produce the right model in Fig. 8b (right version).

```
move target in attack.parameters from 1 to 2
set character.name from "Character" to "Hero"
set troll.name from "Troll" to "Ogre"
create leftGen type Generalization
set leftGen.general to character
set knight.generalization to leftGen
delete cast
delete giant
```

And following list is the derived change events for O_R that are obtained from Bob's and Jane's versions in Fig. 8.

Listing 11: The derived change events (operations) made by Alice to produce the right model in Fig. 8c (right version).

```
move gem in attack.parameters from 0 to 1
set character.name from "Character" to "Hero"
set troll.name from "Troll" to "Orc"
remove smash from knight.operations at 0
add smash to giant.operations at 0
create rightGen type Generalization
```

```
set rightGen.general to character
set mage.generalization to rightGen
remove cast from giant.operations at 1
add cast to mage.operations at 0
```

Real Conflict. In EMF Compare, a conflict occurs between two different operations, o_l and o_r, if each is applied to a same element produce two different eventual states, where ! is the operator for expressing that two operations are in conflict. This conflict is classified REAL by EMF Compare.

$$e_o + o_l \not\equiv e_o + o_r \Rightarrow o_l \; ! \; o_r \tag{2}$$

Pseudo Conflict. A conflict is classified as PSEUDO if the eventual states produced are equivalent.

$$e_o + o_l \equiv e_o + o_r \Rightarrow o_l \; !_p \; o_r \tag{3}$$

Non-applicability. A conflict also occurs when applying operation o_l to element e_o makes o_r inapplicable to element e_o. Therefore, operations o_l and o_r are in conflict.

$$(e_o + o_r \not\equiv e_o) \wedge (e_o + o_l + o_r \equiv e_o + o_l) \Rightarrow o_l \; ! \; o_r \tag{4}$$

Using (2), (3), and (4) and information in Listings 11 and 10, we can identify four conflicts – presented in Table 2 along with their conflicting change events. Conflict EC1 is a pseudo duality conflict since both modify the same class character's feature name resulting the same end states, "Hero" or "Hero". Conflict EC2 is a duality conflict. Applying changing troll's name to "Ogre" and troll's name to "Orc" produces two different states – "Ogre" and "Orc". Conflicts EC3 and EC4 are non-applicability conflicts since if we delete operation cast first then it cannot removed from class giant's operations to be added to mage's operations, and if we delete class giant first then operation smash cannot be added to that class' operations.

Conflict detection in state-based comparison might not be accurate since the derived differences/operations might not reflect the real historical changes of a model. For example, EMF Compare does not detect that Alice and Bob modified the same element – parameter target – as indicated by line 29 in List. 8 and line 35 in List. 9 . Using an LCS algorithm, the derived operations related to the feature parameters of element attack, which if presented as change events, are expressed as [`move target in attack.parameters from 1 to 2`] for Bob's version and [`move gem in attack. parameters from 1 to 2`] for Alice's version. Using (5), both operations are not in conflict since both operations modify two different elements, target and gem. The result is different if we employ change-based approach to detect conflicts using the change event records in Listings 8 and 9. We explain this in Section 6.2.

6.2 Change-based Conflict Detection: EMF Store

Non-commutability. In EMF Store [8] , operations o_l and o_r are in conflict if applying them in different order to a same element, in this case element e_o, produces two different eventual states.

$$e_o + o_l + o_r \not\equiv e_o + o_l + o_r \Rightarrow o_l \; ! \; o_r \tag{5}$$

Pseudo non-commutability. However, after examining the implementation [5], even though the eventual states are equivalent, both operations are still treated in conflict.

$$e_o + o_l + o_r \equiv e_o + o_l + o_r \Rightarrow o_l \; !_p \; o_r \tag{6}$$

Non-applicability. This non-applicability rule is the same with the non-applicability rule in the state-based conflict detection. We present the rule again here.

$$(e_o + o_r \not\equiv e_o) \wedge (e_o + o_l + o_r \equiv e_o + o_l) \Rightarrow o_l \; ! \; o_r \tag{7}$$

Composite. If operation o_l is in conflict with operation o_r where o_r is a member of composite operation co_r then operation o_l is also in conflict with each operation o_n in composite operation co_r.

$$o_l \; ! \; o_r \wedge o_r \in co_r \Rightarrow o_l \; ! \; \forall o_n | o_n \in co_r \tag{8}$$

In change-based conflict detection, all operations applied to a model are already available in change-based persistence, thus the operations do not need to be derived trough a diffing process. The availability of real historical changes can improve the accuracy of change detection since we can identify precisely elements that have been changed. In consequence, the undetected conflict in state-based conflict detection can be detected. For example, in Listing 8 line 29, parameter target has been moved from index 1 to 0, while in Listing 9 line 35, it was moved from index 1 to 2. Since both operations modified the same parameter target, using (5) we can identify that both operations are in conflict; parameter target are at different indexes if both operations are applied in different order, and parameters, the containing feature of target, is an ordered feature (Table 3 id ES1).

The drawback of EMF Store is that it only performs comparison between operations to determine conflicts; it does no take into account the end states of models produced by the operations. In consequence, two operations are in conflict just by modifying a same element regardless of the end states that they produce to the element [5]; there is no classification of conflicts to REAL or PSEUDO conflicts. For example, two operations represented by the two change events in Listing 8 at line 37 and Listing 8 at line 32, that change the same feature name to the same value "Hero", are treated in conflict (Table 3 id ES2).

The inconsideration of eventual states also causes the assignments of generalizations leftGen and rightGen to

[5] https://git.eclipse.org/c/emf-store

Table 2: Conflicting change events identified using EMF Compare based on the case in Fig. 8.

ID	Left Change Events (Bob)	Right Change Events (Alice)	Type
EC1	set character.name from "Character" to "Hero"	set character.name from "Character" to "Hero"	pseudo conflict
EC2	set troll.name from "Troll" to "Ogre"	set troll.name from "Troll" to "Orc"	real conflict
EC3	delete cast	remove cast from giant.operations at 1 add cast to mage.operations at 0	non-applicability
EC4	delete giant	remove smash from knight.operations at 0 add smash to giant.operations at 0	non-applicability

Table 3: Conflicting change events identified using EMF Store in Listings 8 and 9.

ID	Left Change Events (Bob)	Right Change Events (Alice)	Type
ES1	set troll.generalization from null to leftGen	unset troll.generalization from rightGen to null set mage.generalization from null to rightGen	non-commutability, composite
	unset troll.generalization from leftGen to null set knight.generalization from null to leftGen	set troll.generalization from null to rightGen	
ES2	set character.name from "Character" to"Hero"	set character.name from "Character" to "Hero"	pseudo non-commutability
ES3	move target in attack.parameters from 1 to 2	move target in attack.parameters from 1 to 0	non-applicability
ES4	unset cast.name from "cast" to null remove cast from giant.operations at 0 delete cast type Operation unset giant.name from "Giant" to null delete giant	remove cast from giant.operations at 0 add cast to mage.operations at 0	non-applicability, composite
		remove smash from knight.operations at 0 add smash to giant.operations at 1	
ES5	set troll.name from "Troll" to "Ogre"	set troll.name from "Troll" to "Orc"	non-commutability

class troll's feature generalization, in Listings 8 at line 38 and 9 at line 33, to be in conflict with the move operations on the opposite sides (Table 3 id ES1). Setting feature troll's generalization to element leftGen is in conflict with the move composite operation that moves rightGen from troll's generalization to mage's generalization. Using the non-commutability (5) and composite (8) rules, we can detect that executing these operations in different order causes troll's generalization has two different eventual values; troll's generalization is null if the move operation is executed first or leftGen if the set operation is executed first. We can use the same reasoning to explain the conflict between setting feature troll's generalization to element rightGen and the move composite operation that moves leftGen from troll's generalization to knight's generalization.

In state-based conflict detection, the latter case (ES1) is not a conflict since the values of class troll's feature generalization in the Jane's, Bob's, and Alice's versions are indentical – all are null. Thus, there are no *derived* operations that concurrently modify class troll's feature generalization. Conflict ES4 is a non-applicability, composite conflict. Moving element smash from class knight to class giant and moving element cast from class giant to class mage require the deletion of class giant to be executed later in order to be applicable. Conflict ES5 is a non-commutability conflict. The name of class troll have an eventual value "Ogre" or "Orc" depending on the execution order of the conflicting operations.

7 Change-based Conflict Detection: Epsilon CBP

In our conflict detection, we take two strategies from both change and state-based conflict detections to improve the accuracy of our approach. First, we exploit change events to accurately address real historical changes – not derived ones – of models. Second, we also take into account the original and eventual states of models being compared. Two sequences of operations that produce two eventual states that are equal to an original state should not be treated as in conflict. For the latter strategy, the original and eventual states is are already calculated during the construction of the element tree. Since we also record all change events to every element and feature that they affect, we can retrieve all related change events that produce the eventual state of an element or feature.

7.1 Theoretical Approach

Let's say that we have the original state of an element e_o. We also have a set of operations $O_L = \{o_{l1}, o_{l2}\}$ that we apply to e_o that changes its state to element e_l.

$$e_o + o_{l1} + o_{l2} \rightarrow e_l \tag{9}$$

We also have another set of operations $O_R = \{o_{r1},\, o_{r2}\}$ that we apply to e_o that produces element e_r.

$$e_o + o_{r1} + o_{r2} \rightarrow e_r \tag{10}$$

Instead of calculating conflict between operations, we start by checking the equivalency of an element's left and right states to its original state. If the states on both sides are equivalent to the original state, regardless of how many operations have been applied, we can infer that there is no conflict between the members of the two operation sets, O_L and O_R, since there is no change of eventual states. We also identify no conflict if an element is only modified on one side – no operations applied on the other side.

$$\begin{aligned} &e_o \equiv e_l \wedge e_o \equiv e_r \vee |O_L| > 0 \vee |O_R| \Rightarrow \\ &\neg(\forall o_l \;!\; \forall o_r) \mid o_l \in O_L, o_r \in O_R \end{aligned} \tag{11}$$

Therefore, we can also argue that if both states, e_l and e_r, are not equivalent to the original state e_o, and, at least, there is an operation applied to the element on each side then we can conclude that operation set O_L is in conflict with the operation set O_R.

$$\begin{aligned} &e_o \not\equiv e_l \vee e_o \not\equiv e_r \wedge |O_L| > 0 \wedge |O_R| > 0 \Rightarrow \\ &\forall o_l \;!\; \forall o_r \mid o_l \in O_L, o_r \in O_R \end{aligned} \tag{12}$$

As in EMF Compare, we also implement pseudo conflict. Pseudo conflict is a conflict where e_l and e_r are equivalent thus it can be automatically resolved in conflict resolution without user intervention; choosing any side produces the same eventual state.

$$\begin{aligned} &e_o \not\equiv e_l \vee e_o \not\equiv e_r \wedge e_l \equiv e_r \wedge |O_L| > 0 \wedge |O_R| > 0 \\ &\Rightarrow \forall o_l \;!_p\; \forall o_r \mid o_l \in O_L, o_r \in O_R \end{aligned} \tag{13}$$

7.2 Procedure for Detecting Conflicts

We perform procedure in Alg. 4 and employ (11) and (12) inside it to identify conflicts between two CBPs. Basically, the algorithm works by iterating through all the elements, features, and values in the element three (Fig. 9) and checking the equivalency of their original and eventual states as well the numbers of operations applied to them. The results are then used as inputs to decide whether a conflict has been detected or not.

The algorithm starts by creating a empty list conflictList to contain identified conflicts (lines 2). The algorithm then iterates through all the elements, features, and values in the element three. At lines 4 to 11, the algorithm checks if there is a conflict related to a deletion of an element. If an element is deleted on one side or both sides, it means that all events related to the element on both sides

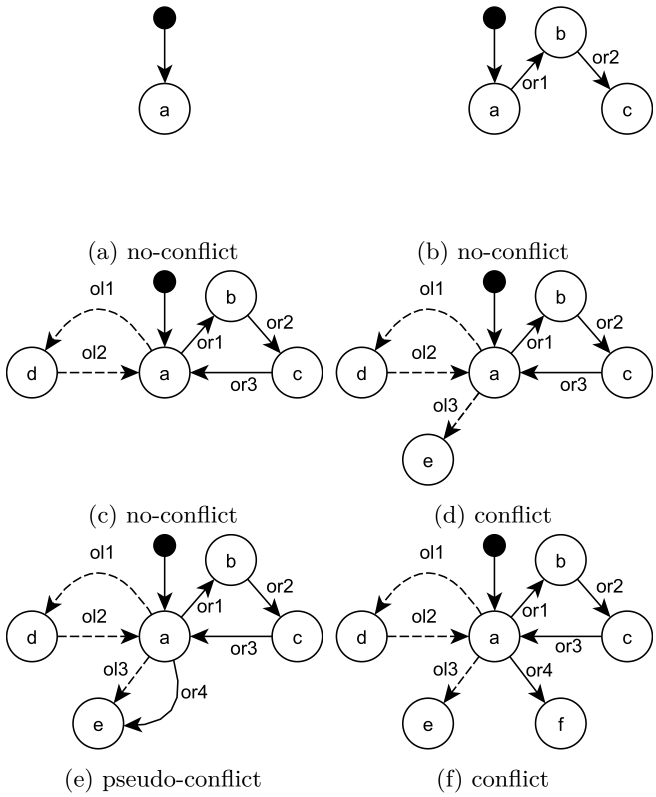


Figure 10: Conflicting and non-conflicting changes.

should be in conflict. To get all the related events, the algorithm use functions getAllRelatedLeftEvents(element) and getAllRelatedLeftEvents(element) (elementact as a map key to access the events) that return two sets of the related events, leftEvents and rightEvents respectively. The related events are events applied to the deleted element, including its sub-elements and features, and events that are parts of composite events. If both sets of events are not empty then a conflict is created containing both sets of events. If the element is deleted on both sides then we set the conflict as PSEUDO. The identified conflict is then added to conflictList.

7.2.1 Conflict with Deletion As an example, when the iteration reach element giant, the algorithm identifies that the element has been deleted only on the left side. The algorithm then collects all the related events from both. The left events consist of all events as parts of the deletion of element giant composite event. The right events consist of events that remove operation smash from class knight and add it to class giant and events that remove operation cast from class giant to class mage. The algorithm then creates an conflict that consists of these events producing conflict CB3 in Table 4. The conflict is not set PSEUDO since the element is only deleted on the left side.

7.2.2 Conflict between Cross-container Moves Lines 15 to 25 are dedicated to identify conflicts related to cross-

Algorithm 4: Algorithm for conflict detection using element tree.

```
input  : an instance of ElementTree elementTree
begin
    conflictList ← ConflictList();
    foreach element in elementTree do
        // Handle conflicts with deletion -------------------
        if isLeftDeleted(element) or isRightDeleted(element) then
            leftEvents ← getAllRelatedLeftEvents(element);
            rightEvents ← getAllRelatedRightEvents(element);
            if size(leftEvents) > 0 and size(rightEvents) > 0 then
                conflict ← createConflict(leftEvents, rightEvents);
                if isLeftDeleted(element) and isRightDeleted(element) then
                    setPseudo(conflict);
                end
                addConflict(conflict, conflictList);
            end
        end
        // Handle conflicts with cross-container move -----------------
        if (getOriginalContainer(element) <> getLeftContainer(element) or getOriginalContainingFeature(element) <>
          getLeftContainingFeature(element)) or (getOriginalContainer(element) <> getRightContainer(element) or
          getOriginalContainingFeature(element) <> getRightContainingFeature(element)) then
            leftEvents ← getAllRelatedLeftEvents(element);
            rightEvents ← getAllRelatedRightEvents(element);
            if size(leftEvents) > 0 and size(rightEvents) > 0 then
                conflict ← createConflict(leftEvents, rightEvents);
                if getLeftContainer(element) = getRightContainer(element) and getLeftContainingFeature(element) =
                  getRightContainingFeature(element then
                    setPseudo(conflict);
                end
                addConflict(conflict, conflictList);
            end
        end
        foreach feature in getFeatures(element) do
            // Handle single-valued feature -----------------
            if isSingleValued(feature) then
                originalValue ← getOriginalValue(feature);
                leftValue ← getLeftValue(feature);
                rightValue ← getRightValue(feature);
                leftEvents ← getAllRelatedLeftEvents(element, feature);
                rightEvents ← getAllRelatedRightEvents(element, feature);
                if originalValue <> leftValue or originalValue <> rightValue and size(leftEvents) > 0 and
                  size(rightEvents) > 0 then
                    conflict ← createConflict(leftEvents, rightEvents);
                    if  leftValue = rightValue then
                        setPseudo(conflict);
                    end
                    addConflict(conflict, conflictList);
                end
            // Handle multi-valued feature -----------------
            else if isMultiValued(feature) then
                if isOrdered(feature) then
                    values ← getUnequalLeftAndRightValues(feature);
                    foreach value in values do
                        leftEvents ← getAllRelatedLeftEvents(element, feature, value);
                        rightEvents ← getAllRelatedRightEvents(element, feature, value);
                        if size(leftEvents) > 0 and size(rightEvents) > 0 then
                            conflict ← createConflict(leftEvents, rightEvents);
                            if getLeftIndex(value, feature) = getRightIndex(rightValue, feature) then
                                setPseudo(conflict);
                            end
                            addConflict(conflict, conflictList);
                        end
                    end
                else if not isOrdered(feature) then
                    values ← getXORLeftAndRightValues(feature);
                    foreach value in values do
                        leftEvents ← getAllRelatedLeftEvents(element, feature, value);
                        rightEvents ← getAllRelatedRightEvents(element, feature, value);
                        if size(leftEvents) > 0 and size(rightEvents) > 0 then
                            conflict ← createConflict(leftEvents, rightEvents);
                            if isExisted(value, feature) = isExisted(rightValue, feature) then
                                setPseudo(conflict);
                            end
                            addConflict(conflict, conflictList);
                        end
                    end
                end
            end
        end
    end
    return conflictList;
end
```

Table 4: Conflicting change events in Listings 8 and 9 identified using the proposed conflict detection.

ID	Left Change Events (Bob)	Right Change Events (Alice)	Type
CB1	set troll.name from "Troll" to "Ogre"	set troll.name from "Troll" to "Orc"	non-commutability
CB2	move target in character.parameters from 1 to 2	move target in character.parameters from 1 to 0	non-commutability
CB3	unset cast.name from "cast" to null remove cast from giant.operations at 0 delete cast type Operation unset giant.name from "Giant" to null delete giant	remove cast from giant.operations at 0 add cast to mage.operations at 0 remove smash from knight.operations at 0 add smash to giant.operations at 1	non-applicability, composite
CB4	set character.name from "Character" to "Hero"	set character.name from "Character" to "Hero"	pseudo, non-commutability

container moves. First, the algorithm checks if an element has been moved from its original container to another container, on one of the sides or both sides. If only on one side or both sides the element has been moved to another container, it continues to check the number of events related to the element by firstly obtaining change events related to the element on both sides using functions getAllRelatedLeftEvents(*element*) and getAllRelatedLeftEvents(*element*) yielding two sets of events, leftEvents and rightEvents. If the element has, at least, one event on each side, a conflict is created containing leftEvents and rightEvents. If on both sides the element is moved to the same container then the conflict is set to PSEUDO. The conflict is then added to conflictList.

7.2.3 Single-valued Feature Conflict Similar procedure also applies to single-valued features (lines 27-39). The procedure starts by retrieving leftValue, rightValue, and originalValue of a single-valued feature. It then checks the inequality of leftValue and rightValue to originalValue. If one of leftValue and rightValue are not equal to originalValue, it then continues to check the number of change events related to the feature by firstly retrieving them using function getAllRelated*Events($element, feature$) (element and feature act as a map key to access the events) yielding two sets of related events, leftEvents and rightEvents. If leftEvents and rightEvents are not empty then a conflict that contains these events is instantiated. The procedure then checks the equality of leftValue and rightValue and set the conflict to PSEUDO if leftValue and rightValue are equal. Finally, the conflict is put into conflictList.

For example, when the iteration reach feature name of class troll, the algorithm retrieves the left, right, and original values of the feature, yielding “Ogre”, “Orc”, and “Troll” respectively. Since “Ogre” and Orc” are not equal to “Troll”, the algorithm continues to retrieve two sets of events related to the feature. Only one event contained exists in each set. On the left side, the event sets the name of class troll from “Troll” to “Ogre”, while on the right side, the the event sets it from “Troll” to “Orc”. Both event sets are not empty thus a conflict containing them is created. Since “Ogre” is not equal to “Orc” the conflict is not set to PSEUDO. This conflict is the conflict CB1 in Table 4. This part of the algorithm also identifies conflict CB4 except that this conflict is set to PSEUDO since both sides change class character’s name to “Hero”.

7.2.4 Ordered Multi-valued Feature Conflict Conflict detection for ordered, multi-valued features (lines 41 to 53) relies on function getUnequalLeftAndRightValues. The function returns all values from left and right sides that are not equal to their original states in terms of (in)existence and indexes. For example, in Fig. 9, parameter target in feature parameters is at index 2 on the left side but at index 1 in its original state. Thus, the value is included in the returned set. On the right side, this parameter is also at different index to its original index but it is already included the returned set.

The algorithm then iterates through the values of the set. For each value, it retrieves all events related to the value on this feature (element, feature, and value act as a map key to access the events) using function getAllRelated*Events($element, feature, value$) yielding two sets of events, leftEvents and rightEvents. If both sets of events are not empty then a conflict is created. If the value on both sides is at the same index then the conflict is PSEUDO. Lastly, the conflict is added to conflictList. The parameter target in feature parameters has been concurrently modified; it has one event on each side: parameter target is moved to the last index on the left side and to the first index on the right. Thus, a conflict in detected. This conflict is presented as conflict CB2 in Table 4.

7.2.5 Unordered Multi-valued Feature Conflict Conflict detection for un-odered, multi-valued features is handled at lines 54 to 67. Instead of using function getUnequalLeftAndRightValues, it employs function getXORLeftAndRightValues. The function also returns all values from left and right sides that are not equal to their original states but only in terms of (in)existence since indexing is not considered in un-ordered features. The procedure to detect

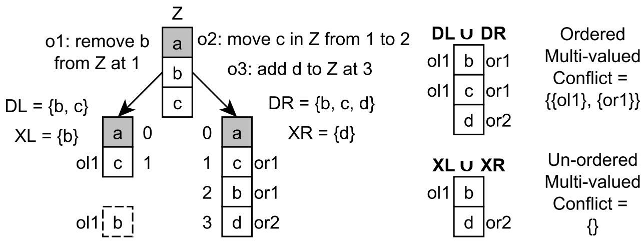


Figure 11: Detecting conflicts in Multi-valued features (D: values that are different at every index; X: XOR of values).

a conflict is similar to the procedure for ordered features except that it check the existence of values (function isExisted) to determine if a conflict is PSEUDO or not.

8 Merging

Conflict resolution and merging strategies are out of the scope of this paper. However, it is important to present the merged models of these three approaches in order inspect their correctness. Thus, we present the merged models of the models in Fig. 8 using these three different approaches. The merge is all-left-to-right merge which means the left changes is prioritised above right changes. In other words, the right changes is applied first to the original model so that the left changes can override the right changes. If there is a conflict then the conflicting right changes are cancelled. In EMF Compare, this cancellation means only the left changes are executed, the right changes are *not executed*, while, in Epsilon CBP and EMF Store, the right changes are *reversed* so that the affected elements are brought back to their original states.

In the default implementation of EMF Compare, the all-left-to-right merge sets the right model as the target and the left model as the source. This means the left changes are not applied to the original model. Instead, they are applied to the right model. Using this strategy, if we resolve all the conflicts in Table 2 using the all-left-to-right merge, we get a merged model as in Fig. 12b. All the right events are not executed. Instead, in the right model, class character's feature name is set to "Hero" and class troll's feature name is set to "Ogre". Also, operation cast and class giant are removed from the right model. The removal of class giant also removes operation smash since the operation is contained by the class in the right model.

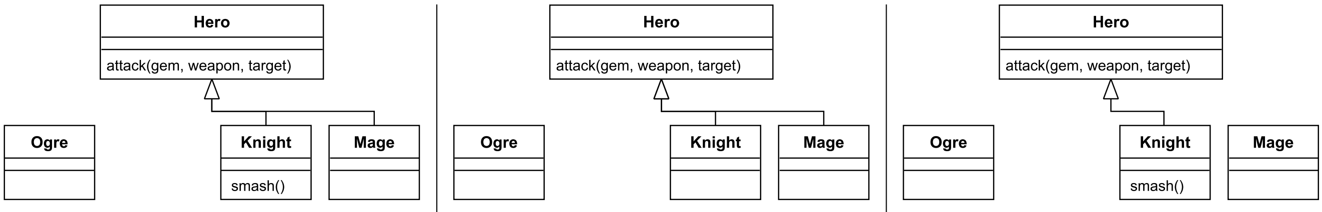
For Epsilon CBP and EMF Store, applying the all-left-to-right merging requires all the right events of the conflicts in Tables 3 and 4 to be reversed. This reversal brings back the states of the elements affected by the conflicting right events to their original states which makes the conflicting left events safe to be applied to the original model. All right events are applied first followed by the reversed conflicting right events and then by all left events (see Listings 12 and 13). In this order, the events are applied to the original model, not the right model as in the EMF Compare, to produce the merged model.

Listing 12: Merged change events (operations) of the models in Fig. 8 and Listings 8 and 9 using EMF Store. The commented lines are added only to improve readability.

```
#--right events--
move target in attack.parameters from 1 to 0
remove smash from knight.operations at 0 composite l1
add smash to giant.operations at 0 composite l1
remove cast from giant.operations at 1 composite l2
add cast to mage.operations at 0 composite l2
create rightGen type Generalization
set rightGen.general to character
set troll.generalization to rightGen
set character.name from "Character" to "Hero"
remove rightGen from troll.generalization composite
    l3
set mage.generalization to rightGen composite l3
set troll.name from "Troll" to "Orc"
#--reversed conflicting right events--
set troll.name from "Orc" to "Troll"
remove rightGen from mage.generalization composite
    l3r
set troll.generalization to rightGen composite l3r
set character.name from "Hero" to "Character"
remove rightGen from troll.generalization
set rightGen.general from character to null
delete rightGen
remove cast from mage.operations at 0 composite l2r
add cast to giant.operations at 1 composite l2r
remove smash from giant.operations at 0 composite l1r
add smash to knight.operations at 0 composite l1r
move target in attack.parameters from 0 to 1
#--left events--
create leftGen type Generalization
set leftGen.general to character
set troll.generalization to leftGen
set character.name from "Character" to "Hero"
remove leftGen from troll.generalization composite r1
set knight.generalization to leftGen composite r1
move target in attack.parameters from 1 to 2
unset cast.name from "cast" to null composite r2
remove cast from giant.operations at 0 composite r2
delete cast composite r2
unset giant.name from "Giant" to null composite r2
delete giant comp r2
set troll.name from "Troll" to "Ogre"
```

Listing 13: Merged change events (operations) of the models in Fig. 8 and Listings 8 and 9 using Epsilon CBP. The commented lines are added only to improve readability.

```
#--right events--
move target in attack.parameters from 1 to 0
remove smash from knight.operations at 0 composite l1
add smash to giant.operations at 0 composite l1
remove cast from giant.operations at 1 composite l2
add cast to mage.operations at 0 composite l2
create rightGen type Generalization
set rightGen.general to character
set troll.generalization to rightGen
set character.name from "Character" to "Hero"
remove rightGen from troll.generalization composite
    l3
set mage.generalization to rightGen composite l3
set troll.name from "Troll" to "Orc"
#--reversed conflicting right events--
set troll.name from "Orc" to "Troll"
remove cast from mage.operations at 0 composite l2r
add cast to giant.operations at 1 composite l2r
remove smash from giant.operations at 0 composite l1r
add smash to knight.operations at 0 composite l1r
```



(a) Epsilon CBP (b) EMF Compare (c) EMF Store

Figure 12: Merged models of models in Fig. 8 after applying all-left-to-right merging.

```
move target in attack.parameters from 0 to 1
#--left events--
create leftGen type Generalization
set leftGen.general to character
set troll.generalization to leftGen
set character.name from "Character" to "Hero"
remove leftGen from troll.generalization composite r1
set knight.generalization to leftGen composite r1
move target in attack.parameters from 1 to 2
unset cast.name from "cast" to null composite r2
remove cast from giant.operations at 0 composite r2
delete cast composite r2
unset giant.name from "Giant" to null composite r2
delete giant comp r2
set troll.name from "Troll" to "Ogre"
```

The drawback of the EMF Compare approach is that it set the right model as the target for merging changes resulting the lost of operation smash in the merged model (Fig. 12b). In contrast, Epsilon CBP and EMF Store reverse the event that moves operation smash from class knight to class giant so that it moves back operation smash from class giant to class knight. The drawback of EMF Store is that it does not concern about the eventual states produced by the conflicting events. It only concern that a feature has been modified concurrently on both sides identified by the presence of, at least, an event on each side. For example, the eventual states of class troll's feature generalization are null in the original, left, and right models. However, these are not considered by EMF Store. Instead, it only identifies that there are events on both sides associated to this feature. Thus, conflict ES1 in 3 is generated. In consequence, all events related to the feature on the right side, and the events that are part of their composite events, are reversed resulting the exclusion of generalization rightGen from the merged model. This drawback has been addressed by the Epsilon CBP by checking the equality of the original, left, and right states of the feature. Since they are all equal then there is no conflict between the events. Thus, Epsilon CBP produces a merged model as in Fig. 12a.

9 Evaluation Method

In this section, we present the method that we employed to evaluate our change-based comparison approach and discuss the results. We also present the limitations and threats to the validity of the evaluation.

In order to assess the performance benefits of the change-based approach in terms of model comparison – differencing and conflict detection, we have evaluated it against a mature and widely-used state-based comparison tool (EMF Compare [6,9]). Since there are no manually developed, large models persisted in our change-based format yet, the dataset for our experiments was constructed from a large model reverse-engineered from the Eclipse Epsilon project [10,11]. This model conforms to the Java metamodel [12] and consists of more than 1.6 million elements with a size of 224 MBs when persisted in XMI.

We cloned the original model to produce two new (left and right) models and perform operations (add, remove, move, set with random elements, features, indexes, and values) on both models to create differences. We made 1.1 million artificial changes to each model, generating over 1.1 million events (one operation can generate more than one event, e.g. a move between features generates remove and add events). Events generated by the changes were persisted in our change-based format (to be used later in change-based model comparison). After every 50,000 changes, we made a measurement point. We persisted the last state of the models in state-based format (to be used later in state-based model comparison) and then performed change-based and state-based model comparison and measured their execution time and memory footprint. We created 22 measurement points to capture their trends in one experiment.

9.1 Model Differencing

We conducted five experiments to evaluate the model differencing of our approach. In the first experiment, the ratio of occurrence between add, remove, move, and set changes is set to 1:1:20:40 intuitively in assumption that in a mature model modification – move and set events – occurs more frequent than addition and deletion. Since we wanted the change of total elements not to affect our measurement, the number of total elements should be kept constant. For example, it is difficult to tell an increase of time in comparison is caused by an increase in the number of elements or by the number of change events. One way to do this was to exclude add and remove operations. However, excluding both operations made measurement less representative. Thus, we still included both operations but made their probabilities equal so that the number of total elements remain largely unchanged.

In the rest of the experiments, we only performed homogeneous type operations – isolated from other types – per experiment (e.g. add-only, move-only operations). In the end, we obtained 5 results of the experiments: mixed, add-only, remove-only, move-only, and set-only measurement results. We did this to asses whether operations of different types have a different impact on model comparison.

For the change-based approach, the comparison time comprises loading change events, constructing an element tree, and identifying differences. The memory footprint is the space used to hold the change events, element tree, and differences in memory. For the state-based approach, the comparison time comprises matching elements and identifying differences, and the memory footprint is the space required to hold the matches and differences in memory. All measurements were performed on the same machine with the following specification: AMD Opteron(tm) Processor 6386 SE @ 2.8 GHz cache size 2 GBs (64 processors), 528 GBs main memory, Ubuntu 16.04.6 LTS operating system, and Java(TM) SE Runtime Environment (build 1.8.0_201-b09) with JVM InitialHeapSize 2GBs and MaxHeapSize 32 GBs.

Since the change-based and state-based approaches can produce a different number of syntactically equivalent differences, in order to evaluate the correctness of the change-based approach, we reconciled all the differences by performing all-left-to-right merging – making the right model identical to the left model – based on the identified differences. If the all-left-to-right merging of change-based approach produces a model that is identical to the model produced by the all-left-to-right merging of the state-based approach then it can be said that differences identified by the change-based approach are correct. We performed this correctness checking at every measurement point.

9.2 Conflict Detection

In evaluating our conflict detection approach, basically we followed the similar procedures as in the model differencing evaluation, except that we add another implementation of change-based model persistence (EMF Store [8]) to compare it with our approach. We did not include it in the model differencing evaluation since it works purely on operations, and it is designed to identify conflict between operations; not for finding differences between models. We imported the changes persisted in our change-based format into EMF Store by replaying the changes in the EMF Store. Thus, we could obtain equivalent changes but in the EMF Store format. Since in this evaluation we used two change-based persistence: our approach and EMF Store, we use the term Epsilon CBP to refer to our approach. Due to slow execution of replaying delete event in EMF Store, we have to reduce the size of the models to 70 thousand elements each and the number of changes to 550 thousands with 25 thousand changes for each measurement point – 22 measurement points in total.

10 Evaluation Results and Discussion

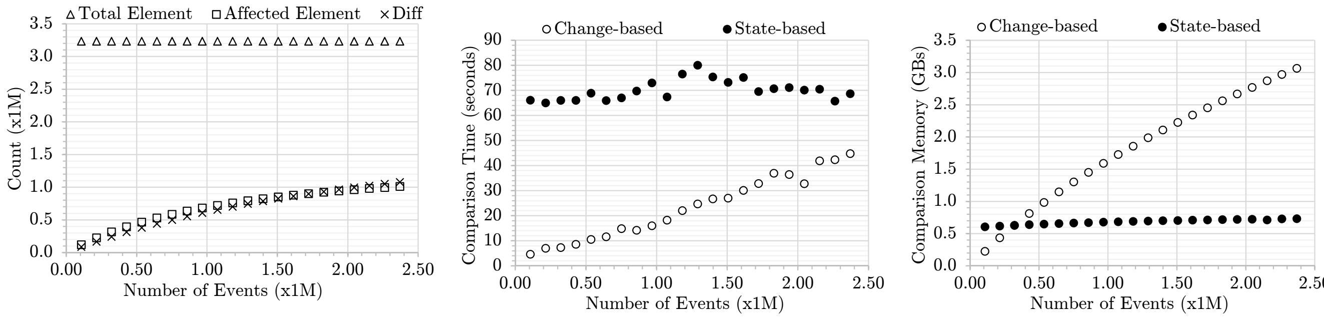
In this section, we report on the obtained results in terms of comparison time and memory footprint for both model differencing and conflict detection.

10.1 Model Differencing

This section presents the results of both mixed and homogeneous operation measurements for the model differencing evaluation.

10.1.1 Mixed Operations In the mixed operation measurement, we modify two identical models differently by applying random operations. As the number of change events generated by the modification grows, the numbers of affected elements and differences also increase in a logarithmic manner. The patterns can be seen in Fig. 13a. The growth is logarithmic since the probability that the random operations modify the same elements also increases. Thus, some change events might not contribute to the addition of new affected elements and differences. In other words, more events are required to increase the number of affected elements or differences. In Fig. 13a, the total elements remains largely unchanged due to the equal probabilities of addition and deletion as has been set in Section 9. The figure gives us an insight about the characteristics of the modification caused by the random operations in the mixed operation measurement; it supports explaining the implication of the changes on execution time and memory footprints of model comparison.

After applying some random changes on both models, the modification produces 100,000 change events at the first measurement point. Using this amount of events, our change-based comparison only takes 5 seconds to identify around 90,000 differences, in contrast to the state-based comparison that takes 66 seconds (see the first measurement points in Figures 13a and 13b). If the modification continues, more changes events are generated. This growing number of change events has to be loaded into memory and thus slows down the change-based comparison. Nevertheless, the change-based comparison is still faster than the state-based comparison even though the number of change events reaches 2.37 millions – more than 1 million differences at that point; the change-based comparison outperforms the state-based comparison in execution time (Figure 13b). Fig. 14a breaks down the comparison time in detail. It exhibits that the event loading time is the dominant contributor to the slowdown

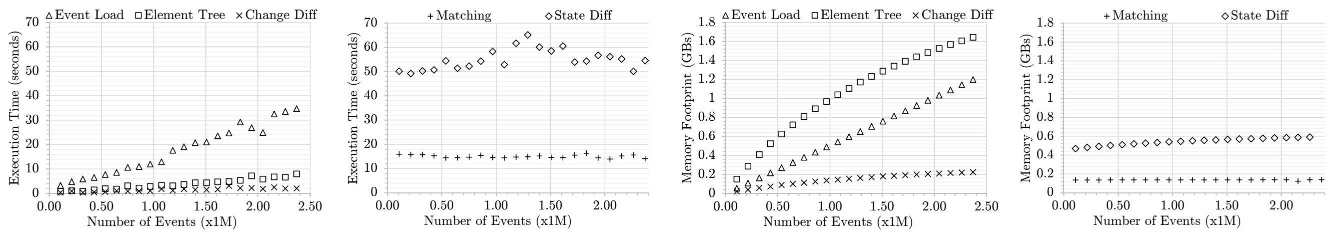


(a) total elements, affected elements, and diffs

(b) execution time

(c) memory footprint

Figure 13: Change-based vs. state-based model comparison as change events increase.



(a) change-based comparison time

(b) state-based comparison time

(c) change-based memory footprint

(d) state-based memory footprint

Figure 14: Breakdown view of comparison time and memory footprint in Figure 13.

compared to the element tree's construction time and diffing time.

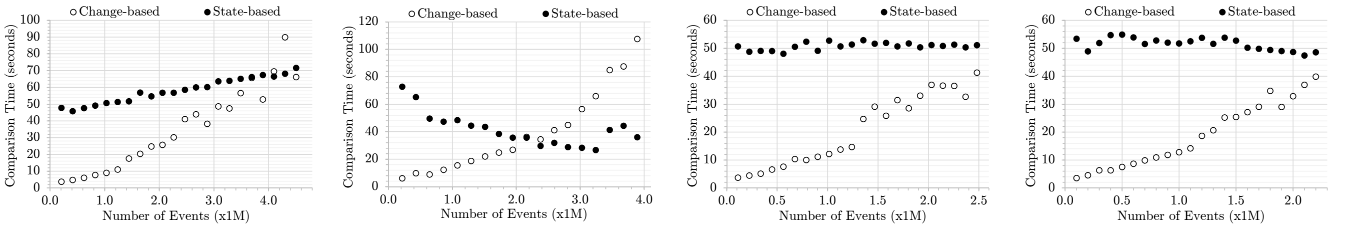
For the state-based comparison in Fig. 14b, the comparison time only experiences a slight increase as the number of identified differences also grows. This slight increase is contributed mainly by the diffing time, while the matching time tends to be constant due to the very small increase of the total elements (Figures 13a).

Nevertheless, change-based comparison generally consumes more memory than the state-based comparison (see Figure 13c). It only consumes less memory than its state-based counterpart when the number of events is less than 0.3 millions (around less than 0.25 million identified differences at that moment). Fig. 14c breaks down the memory footprint of change-based comparison into three factors: the loaded change events, element tree, and diffs. As modification continues, an increasing number of events is generated. These events have to be loaded into memory since they contain the required information for the construction of an element tree. The amount of space to keep these change events in memory grows linearly with their number.

In contrast, the memory used for the element tree grows logarithmically. As the number of events increases, the probability that events modify already affected elements also increases. Thus, no additional memory allocation is required for the element tree. We can also notice that the element tree occupies most of the memory footprint since it mirrors the partial states – elements, features, and values – of the models that are affected by the changes. Moreover, in our technical implementation, a feature can have many instances – one instance for each element (As a comparison, in the EMF implementation, there is only one instance for a feature. The feature is used as a key so that different elements can have the same feature that maps to different values simultaneously). This contributes to the large memory footprint used by the element three. The identified change-based diffs, the third factor, are the smallest factor that contributes to the memory footprint of the change-based comparison.

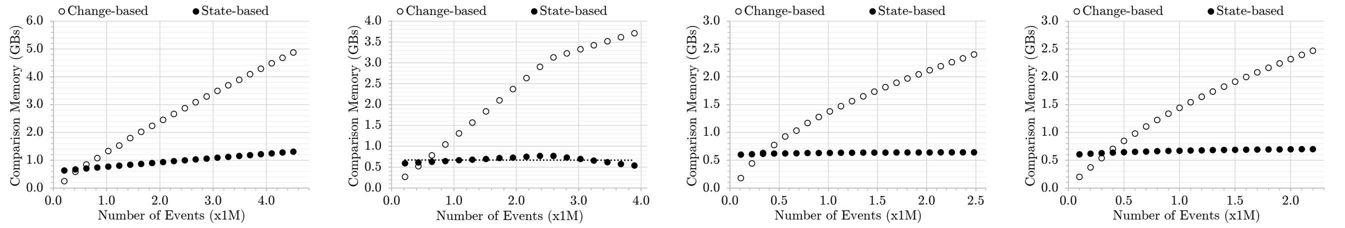
For the state-based comparison in Fig. 14d, the memory footprint only grows slightly along the increase of differences. A large part of the memory footprint is used to represent the identified differences, while the memory used for matches tends to be constant as the changes of the total elements are very small – less new elements means less memory needs to be allocated for new matches (Figures 13a).

10.1.2 Homogeneous Operations Figures 15 and 16 exhibit the comparison time and memory footprint of models that have been modified using homogeneous operations – `add`, `remove`, `move`, or `set` only. We can notice that in all figures change-based comparison outperforms its state-based counterpart particularly when the num-



(a) add-only (b) delete-only (c) move-only (d) change-only

Figure 15: Comparison time for homogeneous operations.



(a) add-only (b) delete-only (c) move-only (d) change-only

Figure 16: Memory footprint for homogeneous operations.

ber of change events is small relative to the size of the model. As the number of modifications grows, eventually change-based comparison becomes slower than state-based comparison. In our experiments, this happens when the number of events is greater than 4 million (Fig. 15a). Change-based comparison also becomes slower when the size of models shrinks (due to a large number of delete events) as depicted in Fig. 16b as the change-based comparison still needs to load these change events and construct its element tree; in contrast, deletion means less work for state-based comparison. In terms of memory footprint, change-based comparison only performs better than state-based comparison when the number of change events is less than 0.3 millions as depicted in Fig. 16.

10.2 Conflict Detection

This section presents the results of the conflict detection evaluation.

10.2.1 Mixed Operations Similar to the results in the model differencing evaluation (Fig. 13a), the growing number of change events in the conflict detection evaluation is also followed by the logarithmic increase of affected elements (Fig. 17a). The total number of both elements can also be kept relatively constant due to 1:1 ratio of `add` and `delete` operations' occurrence. These change events produce different numbers of conflicts for Epsilon CBP, EMF Compare, and EMF Store as can be seen in Fig. 17b. However, the numbers of conflicts detected are slightly different due to their distinct conflict detection approaches.

Fig. 17c exhibits Epsilon CBP outperforms EMF Compare and EMF Store in terms of execution time in detecting conflicts. In a comparison of two models that involves 1.01 million elements in total and 1.3 millions change events (the last measurement point), Epsilon CBP takes only 6.7 seconds to detect all conflicts while EMF Compare requires 21.4 seconds to finish the conflict detection. Both conflict detections are faster than EMF Store that needs 1 minute and 34.6 seconds to complete detecting conflicts. Fig. 17d also shows Epsilon CBP outmatches EMF Compare and EMF Store in terms of memory footprint in conflict detection. At the last measurement point, Epsilon CBP only consumes 5 GBs which is much lesser than EMF Compare and EMF Store that occupy around 16 and 26 GBs of memory footprint respectively.

Fig. 18 shows the detailed view of Epsilon CBP, EMF Compare, and EMF Store on the time required to complete conflict detection. As can be seen in the Fig. 18a, Epsilon CBP takes most of the time used to load change events and construct element tree compared to the time it takes for identifying conflicts. In detecting conflicts, the Epsilon CBP does not requires to perform differencing since changes are already available in the form of change events Thus, the differencing is not included in the diagram. In EMF Compare, at the last measurement point, we can notice that the time taken for matching and identifying conflicts is less than 6 seconds each, which is smaller than the time used for identifying differences that is 12.6 seconds (Fig. 18b). The differencing takes a great portion of the time since it needs to derive differences twice; differences between left and original models and right and original models. The time for for matching and

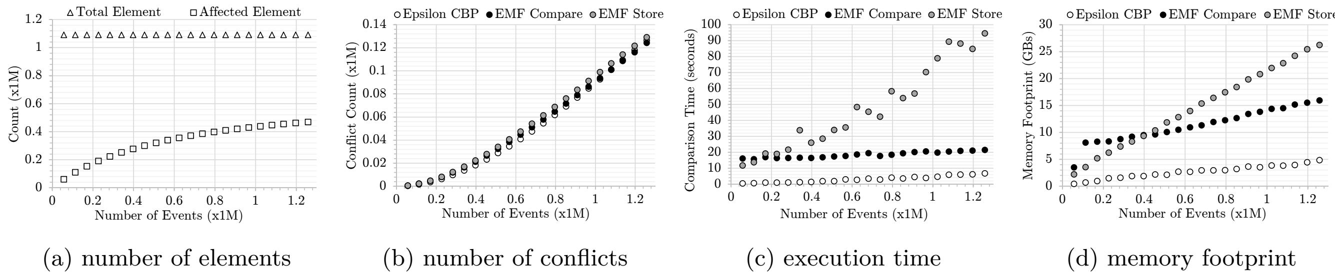


Figure 17: Epsilon CBP vs. EMF Compare vs. EMF Store comparison as change events increase.

differencing tends to have only slight increase since the sizes of the models are set to be as constant as possible (Fig. 13a). In contrast, the time for detecting conflicts tends to grow faster than both due to the increasing number of conflicting (derived) changes as the number of modification – change events – increases. In detecting conflicts, EMF Store allocates more than a half of the consumed time for identifying conflicts. The rest of the time is used for loading changes and mapping between affected elements and their ids (Fig. 18c).

In terms of memory footprint, Epsilon CBP allocates most of the memory space for element tree construction and the rest is for the loading change events and identifying conflicts (Fig. 19a). The reason for this is due to our technical implementation explained in Section 10.1.1. In EMF Compare (Fig. 19b)), the amount of memory used for matching and differencing only increases slightly due the sizes of the models that are set to be as constant as possible (Fig. 13a). In contrast, the memory used for detecting conflict increases positively as the number of detected conflicts rises (Fig. 17b). For EMF Store, the amount of memory used for loading changes and mapping is slightly above the amount of memory for identifying differences (Fig. 19c).

10.2.2 Homogenous Operations Based on the findings in model differencing and conflict detection evaluation, we argue that the change-based comparison approach works at its best for large models that have been modified a moderate number of times. Models that have been excessively modified and experience significant reduction on model size could impair the performance of change-based comparison as a great number of change records have to be read and loaded into memory.

10.3 Limitations and Validity

The evaluation of the proposed change-based comparison is limited to the Java metamodel only. Thus, there is no guarantee it will perform in a consistent manner on models conforming to different metamodels. Although, we have tried to cover as much as common changes made in EMF models (e.g. performing `add`/`remove`/`set`/`move` operations on `single`/`multi-valued` features, `attribute`/`reference` features, or `containment`/`non-containment` references), the random modification made in the evaluation does not largely reflect the evolution of models in the real world. This is challenging as different domains can have their own patterns of model evolution – different problems, metamodels, modellers, etc.

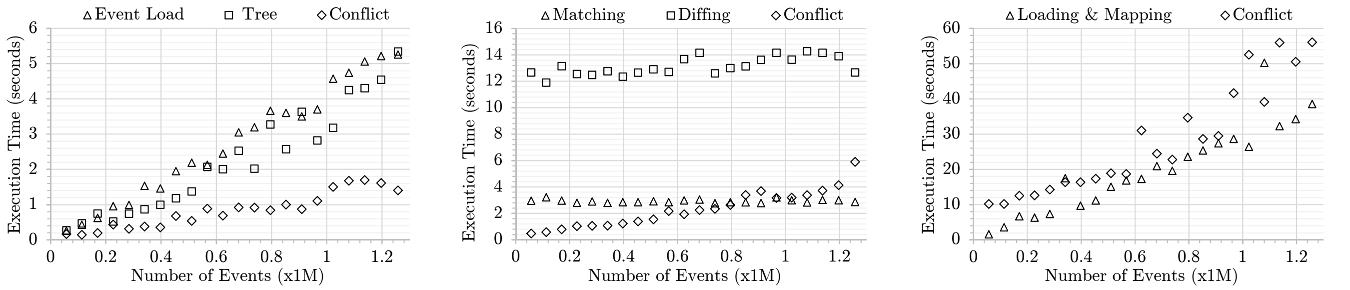
11 Related Work

We are not aware of any other work that targets comparison and diffing of change-based models persisted as files. There are however several existing tools for state-based model comparison. Beyond EMFCompare, which we used for our comparative evaluation due to its maturity and ongoing development activity, tools such as SiDiff [13] and DSMDiff [14] also provide language-agnostic graph-based model comparison, with some room for configuration (e.g. assigning different weights to features of types in the language). Additional expressive power – at the cost of increased complexity and configuration effort – is offered by dedicated comparison languages such as the Epsilon Comparison Language, which can be used to compare both homogeneous and heterogeneous models [15]. We refrain from a more detailed discussion on state-based comparison tools as they all require upfront loading of both versions of the model into memory, which is the main cost that we aspire to reduce with the presented change-based approach.

Database-backed model persistence and version control solutions such as CDO [16], EMFStore [8] also provide diffing capabilities between different versions of the same model without requiring models to be fully loaded into memory, however they present integration challenges with mainstream software engineering tools (e.g. continuous integration systems, backup and restore facilities) which are typically file-based, and their performance can degrade as more models/users are added to a repository, since all models are effectively stored in a single database [17].

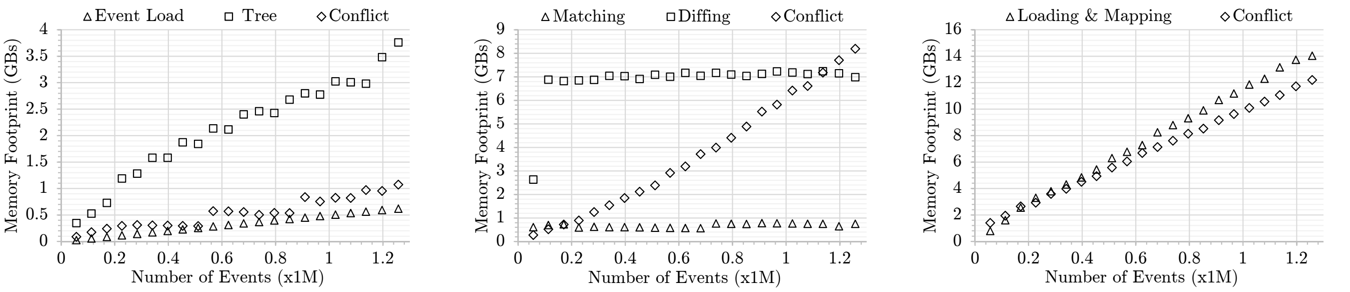
12 Conclusions and Future Work

In this paper, we have presented a novel approach to model comparison by exploiting the nature of change-based persistence which allows us to find differences between versions of a model by only comparing the last



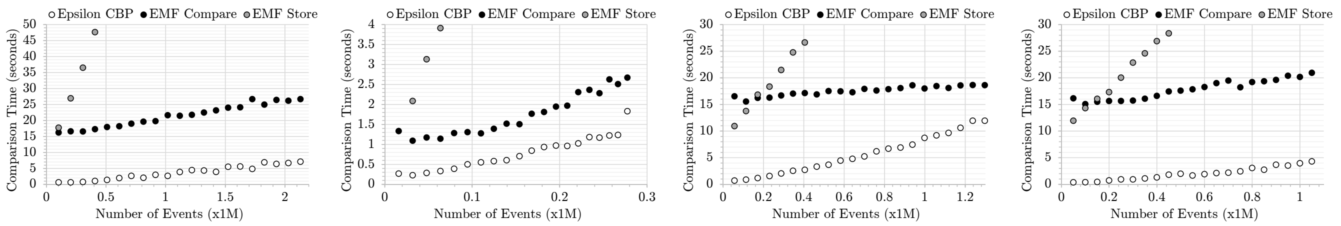
(a) Epsilon CBP (b) EMF Compare (c) EMF Store

Figure 18: Detailed view of Epsilon CBP vs. EMF Compare vs. EMF Store on the time required for conflict detection.



(a) Epsilon CBP (b) EMF Compare (c) EMF Store

Figure 19: Detailed view of Epsilon CBP vs. EMF Compare vs. EMF Store on the memory footprint for conflict detection.



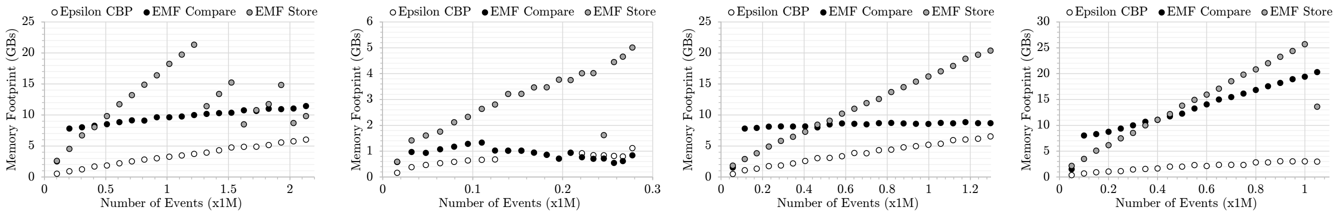
(a) add-only (b) delete-only (c) move-only (d) change-only

Figure 20: Conflict detection time for homogeneous operations.

set of changes between the source and reference model. Our evaluation results suggest that using this approach, we can produce model comparison that is faster than traditional, state-based model comparison. However, the change-based comparison approach needs to load change events from a change-based persistence into main memory and thus may requires more memory than for state-based comparison. In our evaluation, this occurs when the number of change events exceeds 400,000. Arguably, diff and merge operations are usually performed on smaller deltas than our evaluation. The next challenge for future work is to identify strategies to merge models optimally and persist the merging in the change-based way.

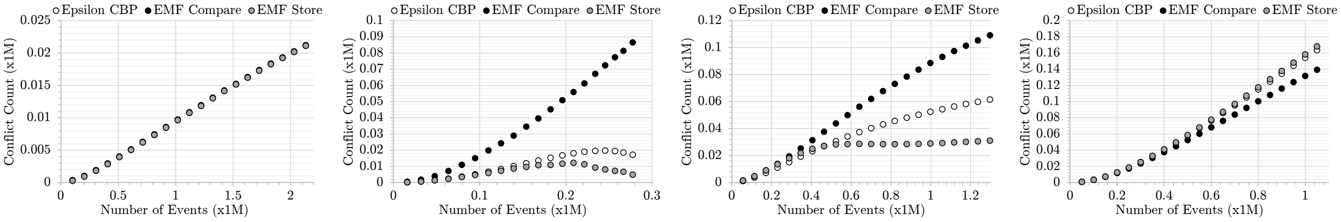
References

1. A. Yohannis, D. S. Kolovos, and F. Polack, "Turning models inside out," in *Proceedings of MODELS 2017 Satellite Events co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017.*, 2017, pp. 430–434. [Online]. Available: http://ceur-ws.org/Vol-2019/flexmde_8.pdf
2. A. Yohannis, H. H. Rodriguez, F. Polack, and D. S. Kolovos, "Towards efficient loading of change-based models," in *Modelling Foundations and Applications - 14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Toulouse, France, June 26-28, 2018, Proceedings*, 2018, pp. 235–250. [Online]. Available: https://doi.org/10.1007/978-3-319-92997-2_15
3. ——, "Towards hybrid model persistence," in *Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE*



(a) add-only (b) delete-only (c) move-only (d) change-only

Figure 21: Conflict detection memory for homogeneous operations.



(a) add-only (b) delete-only (c) move-only (d) change-only

Figure 22: Conflict detection count for homogeneous operations.

21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018., 2018, pp. 594–603. [Online]. Available: http://ceur-ws.org/Vol-2245/me_paper_3.pdf
4. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*, ser. Eclipse Series. Pearson Education, 2008. [Online]. Available: https://books.google.co.uk/books?id=sA0zOZuDXhgC
5. P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, "An introduction to model versioning," in *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, 2012, pp. 336–398. [Online]. Available: https://doi.org/10.1007/978-3-642-30982-3_10
6. EMFCompare, "Emf compare developer guide," https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html, accessed: 2018-11-01.
7. E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986. [Online]. Available: https://doi.org/10.1007/BF01840446
8. M. Koegel and J. Helming, "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, pp. 307–308. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810364
9. Eclipse, "EMF Compare," https://www.eclipse.org/emf/compare/, accessed: 2018-01-15.
10. ——, "Epsilon Git," http://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/commit/?id=ebd0991c279a1f0df1acb529367d2ace5254fe87, accessed: 2018-02-19.
11. ——, "Epsilon," https://www.eclipse.org/epsilon/, accessed: 2018-02-12.
12. ——, "Java Metamodel," https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.java.doc%2Fmediawiki%2Fjava_metamodel%2Fuser.html, accessed: 2019-01-08.
13. C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "Difference computation of large models," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 295–304. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287665
14. Y. Lin, J. Gray, and F. Jouault, "Dsmdiff: a differentiation tool for domain-specific models," *European Journal of Information Systems*, vol. 16, no. 4, pp. 349–361, 2007. [Online]. Available: https://doi.org/10.1057/palgrave.ejis.3000685
15. D. S. Kolovos, "Establishing correspondences between models with the epsilon comparison language," in *Model Driven Architecture - Foundations and Applications*, R. F. Paige, A. Hartman, and A. Rensink, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 146–157.
16. Eclipse, "Eclipse CDO The Model Repository," https://www.eclipse.org/cdo/documentation/, accessed: 2019-04-02.
17. D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot, "A research roadmap towards achieving scalability in model driven engineering," in *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013*, 2013, p. 2.