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

Towards Efficient Conflict Detection of Change-Based Models

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 Conflict detection of two versions of a large model in traditional state-based model comparison can be time-consuming since changes of both models have to be derived first by differencing them to their common ancestor version. This differencing requires every element of both versions to be inspected, matched, and diffed with its respective element in their common ancestor. This can result in bottlenecks in collaborative modelling environments, where detecting conflicts 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 conflict detection. This paper presents how change-based persistence can be used to localise conflict detection so that only elements affected by recent changes that are handled. This approach can substantially reduce conflict detection time, up to 90% in some experiments, compared to state-based model conflict detection.

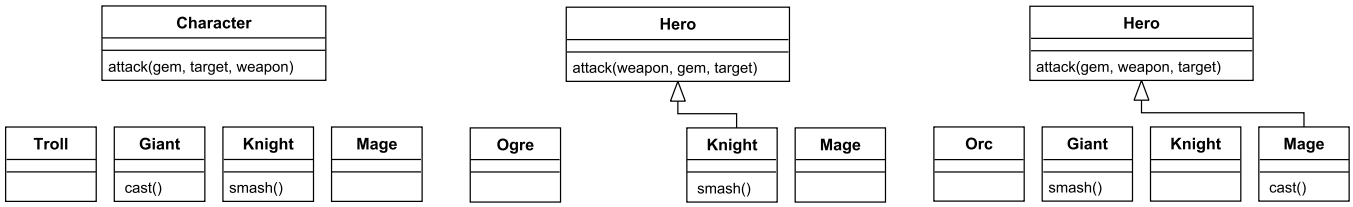
1 Introduction

In the context of Model-Driven Engineering, most of the models are persisted in state-based formats. In such approaches, model files contain snapshots of the models' contents, and activities like version control and change detection are left to external systems such as file-based version-control systems and model differencing facilities. Activities such as model differencing, identifying parts of two versions of a model that are different, and conflict detection, finding conflicting changes between two versions of a model, are computationally consuming for state-based models [1] since every element has to be inspected, matched to its respective pair, and compared to identify their differences, or checked whether changes between versions are in conflict or not [2]. Large parts of these elements might not have been affected by recent changes, particularly in mature models where they are large in sizes but only experience many small fine-tune changes [3]. Thus, inspecting all of the elements is regarded inefficient, and a new approach is required to make the computation more efficient.

As an alternative to state-based persistence, a model can also be persisted in a change-based format, which persists the full sequence of *changes* made to the model instead. The concept of change-based persistence is not new and has been used in persisting changes to software, object-oriented databases, and hierarchical documents [4–6]. The change-based approach can improve detecting differences more precisely at the semantic level, that is by providing finer-granularity information (e.g. types of changes, the order of the changes, elements that were changed, previous values, etc.) and therefore provide support to resolve them [7]. The ordered nature of change-based persistence also means that changes made to a model since its previous version can be identified sequentially without having to inspect every element of the model and compare the model to its previous version. Based on these arguments, This work aims to answer the following research question, **"How and to what extent does change-based model persistence speed up model conflict detection with less side-effect on memory footprint compared to state-based model conflict detection?"** This work has built an implementation of change-based persistence for models conforming to 3-layer metamodelling architectures such as EMF and MOF, and has been using it to speed up model conflict detection. The implementation of the change-based model persistence, the approach that leverages it to speed up model conflict detection as well as its evaluation are presented in this paper.

This paper extends two of our previous work. The first work presents the initial implementation of change-based model persistence [8], and the second work proposes an approach on how to harness change-based model persis-

Send offprint requests to:



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

Fig. 1: Three class diagrams of a Role Playing Game.

tence to speed up model differencing [9]. We extend our previous work by adding the use of change-based model persistence to reduce the time required for model conflict detection. This paper is structured as follows. Section 1 presents the background and motivation of this work. Section 2 introduces the running example that is used throughout this paper to help us explain our proposed approach. Section 3 describes briefly the design and implementation of our change-based model persistence. Section 4 discusses how current approaches, EMF Compare and EMF Store, performs conflict detection. Section 5 describes in detail how our approach exploits change-based model persistence to speed up model conflict detection. Section 6 describes the evaluation that we employed to evaluate our approach. Section 7 discusses the results of the evaluation. Section 8 presents the related work of this study. Finally, Section 9 ends this paper with presenting conclusions and future work of this study.

2 Running Example

Before going deeper into how the change-based persistence is implemented, this paper introduces a running example. Figure 1 shows us three versions of an UML2 model. However, its metamodel has been modified so that a class can only have one superclass as in Java. This example is used throughout this thesis to explain the solutions proposed in this study as well as to explain how conflict detection are performed in the related work, such as in EMF Compare [2] and EMF Store [10].

Let's say that there is a project to develop a simplified class diagram model of a Role Playing Game (RPG). Jane, as the technical leader, set up the initial model (Figure 1a). She then assigned this work to Bob and Alice. Both of them continued to work on the model and made some modification producing the models in Figures 1b and 1c respectively. Persisting these models in the standard XMI [11] format produces three files as shown in Listings 1, 2, and 3.

Listing 1: Simplified XMI file of the original version in Figure 1a.

```
1  <uml:Model>
2  <packagedElement type=Class id="character" name="
       Character">
3  <operation id="attack" name="attack">
4  <parameter id="gem" name="gem"/>
5  <parameter id="target" name="target"/>
6  <parameter id="weapon" name="weapon"/>
7  </operation>
8  </packagedElement>
9  <packagedElement type=Class id="troll" name="Troll"/>
10 <packagedElement type=Class id="giant" name="Giant">
11 <operation id="cast" name="cast"/>
12 </packagedElement>
13 <packagedElement type=Class id="knight" name="Knight
       ">
14 <operation id="smash" name="smash"/>
15 </packagedElement>
16 <packagedElement type=Class id="mage" name="Mage"/>
17 </uml:Model>
```

Listing 2: Simplified XMI file of the left version in Figure 1b.

```
1  <uml:Model>
2  <packagedElement type=Class id="character" name="Hero
       ">
3  <operation id="attack" name="attack">
4  <parameter id="weapon" name="weapon"/>
5  <parameter id="gem" name="gem"/>
6  <parameter id="target" name="target"/>
7  </operation>
8  </packagedElement>
9  <packagedElement type=Class id="troll" name="Ogre"/>
10 <packagedElement type=Class id="knight" name="Knight
       ">
11 <generalization id="leftGen" general="character"/>
12 <operation id="smash" name="smash"/>
13 </packagedElement>
14 <packagedElement type=Class id="mage" name="Mage"/>
15 </uml:Model>
```

Listing 3: Simplified XMI file of the right version of Figure 1c.

```
1  <uml:Model>
2  <packagedElement type=Class id="character" name="
       Character">
3  <operation id="attack" name="attack">
4  <parameter id="gem" name="gem"/>
5  <parameter id="weapon" name="weapon"/>
6  <parameter id="target" name="target"/>
7  </operation>
8  </packagedElement>
9  <packagedElement type=Class id="troll" name="Orc"/>
10 <packagedElement type=Class id="giant" name="Giant">
11 <operation id="smash" name="smash"/>
12 </packagedElement>
13 <packagedElement type=Class id="knight" name="Knight
       "/>
14 <packagedElement type=Class id="mage" name="Mage">
15 <generalization id="rightGen" general="character"/>
16 <operation id="cast" name="cast"/>
17 </packagedElement>
18 </uml:Model>
```

The company where Jane, Bob, and Alice affiliated works with large models and most of the time its employees work in parallel producing different versions of models.

Somehow, as the models grow large, they face bottleneck in their productivity due to slowdown in executing model differencing and conflict detection. They decided to implement the change-based proposed in this paper to speed up the differencing and conflict detection. So, instead of only persisting the snapshots of models, they also persist the complete history of changes of models into change-based model persistence.

Listing 4: Change-based representation of the original version in Figure 1a.

```
1  session "Jane-01"
2  create character type Class
3  set character.name from null to "Character"
4  create attack type Operation
5  set attack.name from null to "attack"
6  add attack to character.operations at 0
7  create gem type Parameter
8  set gem.name from null to "gem"
9  add gem to attack.parameters at 0
10 create target type Parameter
11 set target.name from null to "target"
12 add target to attack.parameters at 1
13 create weapon type Parameter
14 set weapon.name from null to "weapon"
15 add weapon to attack.parameters at 2
16 create troll type Class
17 set troll.name from null to "Troll"
18 create giant type class
19 set giant.name from null to "Giant"
20 create cast type Operation
21 set cast.name from null to "smash"
22 add cast to giant.operations at 0
23 create knight type Class
24 set knight.name from null to "Knight"
25 create smash type Operation
26 set smash.name from null to "smash"
27 add smash to knight.operations at 0
28 create mage type Class
29 set mage.name from null to "Mage"
```

As an example, the complete history of changes made by Jane to construct the original version in Figure 1a is also persisted in a change-based model representation as in Listing 4. The change events (Listing 5) made by Bob is appended to Jane's original change events. Thus, the change events that represent Bob's version (Figure 1b) consists of the original change events and the change events (Listing 5) made by him; only the appended changes are presented on the list. The change events that represents Alice's version (Figure 1c) is presented in Listing 6. One clear advantage of change-based model persistence is that, from Listing 5, we can immediately know all the changes made my Bob and Alice (starting from line 30) and identify all the elements that have been modified since Jane's version.

Listing 5: The appended events made by Bob to produce the left version in Figure 1b (left version).

```
30 session "Bob-01"
31 create leftGen type Generalization
32 set leftGen.general to character
33 set troll.generalization to leftGen
34 set character.name from "Character" to "Hero"
35 unset troll.generalization from leftGen to null
       composite l1
36 set knight.generalization to leftGen composite l1
37 move target in attack.parameters from 1 to 2
38 unset cast.name from "cast" to null composite l2
39 remove cast from giant.operations at 0 composite l2
40 delete cast composite l2
41 unset giant.name from "Giant" to null composite l2
42 delete giant composite l2
43 set troll.name from "Troll" to "Ogre"
```

Listing 6: The appended events made by Alice to produce the right version in Figure 1c (right version).

```
30 session "Alice-01"
31 move target in attack.parameters from 1 to 0
32 remove smash from knight.operations at 0 composite r1
33 add smash to giant.operations at 0 composite r1
34 remove cast from giant.operations at 1 composite r2
35 add cast to mage.operations at 0 composite r2
36 create rightGen type Generalization
37 set rightGen.general to character
38 set troll.generalization to rightGen
39 set character.name from "Character" to "Hero"
40 unset troll.generalization from rightGen to null
       composite r3
41 set mage.generalization to rightGen composite r3
42 set troll.name from "Troll" to "Orc"
```

Let's say the complete scenario that produces the models in Figures 1a, 1b, and 1c as wells Listings 4, 5, and 6 occured according to the following story.

Jane, as the technical leader, set up the initial model. The records of events during setting up the initial is recorded in the CBMP in List. 4. She created a class Character that contains an operation attack with three parameters: gem, target, and weapon (lines 2-15). She also created four other classes; Troll (lines 16-17), Giant (lines 18-22), Knight (lines 23-27), and Mage (lines 28-29). 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 Figure 1a.

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. 6, line 31). She also moved operation smash from class Knight to class Giant and operation cast from class Giant to class Mage as they are more reasonable to belong to their new classes (lines 32-35). Alice also created a generalisation relationship with id rightGen from class Troll to class Character (36-39). Bob also did the same thing except that his generalisation came with id leftGen (List. 5, line 31-33).

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 39). Again, Bob did the same thing. He also changed the name of class Character from "Character" to "Hero" (line 34). 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 40-41), and Bob move generalisation leftGen from class Troll to class Knight (lines 35-36). Bob also moved parameter target in operation attack to the last index as he thought setting target as the last parameter was intuitive (line 37), and deleted the class Giant, and unfortunately, he deleted class Giant accidentally (lines 38-42). The class diagrams of Bob and Alice's models are visualised in Figures 1b and 1c respectively. Lastly, Alice changed the name of class Troll to "Orc" (line 42) while Bob changed it to "Ogre" (line 43).

In Listings 6 and 5, we also introduce composite events – lines with keyword composite – that represent composite change events. Composite change events are events that should be treated as one composition – identified with the same composite id. For example, moving an element from a container to another container is a composite event since it consists of two change events: removing/unsetting the element from its source container and adding/setting it to its target container (lines 40-41 Listing 6).

3 EMF Change-Based Persistence (EMF CBP)

To illustrate the proposed approach, Listing 2 shows a state-based representation of the model of Figure 1b in (simplified) XMI, and Listing 5 shows the proposed equivalent change-based representation of the same model. Instead of persisting a snapshot of the model's state, the representation of Listing 5 captures the complete sequence of change events (create/set/add/move/remove/delete) that were performed on the model since its creation, organised in editing sessions. There are two editing sessions in the case of this model: one in Listing 5 and another in Listing 4. In the actual implementation, change events in Listing 5 are appended to a file that is a copy of Listing 4. Listing 5 only shows the appended change events, while Listing 7 shows the complete change events. Replaying these changes, the change events in Listing 4 first and then followed by the change events in Listing 5, produces the same state as the one captured in Listing 2. Thus, we can conclude that the proposed change-based representation carries at least as much information as the state-based representation.

Listing 7: The complete version of Bob's change events in Listing 5.

```
1  session "Jane-01"
2  create character type Class
3  set character.name from null to "Character"
4  create attack type Operation
5  set attack.name from null to "attack"
6  add attack to character.operations at 0
7  create gem type Parameter
8  set gem.name from null to "gem"
9  add gem to attack.parameters at 0
10 create target type Parameter
11 set target.name from null to "target"
12 add target to attack.parameters at 1
13 create weapon type Parameter
14 set weapon.name from null to "weapon"
15 add weapon to attack.parameters at 2
16 create troll type Class
17 set troll.name from null to "Troll"
18 create giant type class
19 set giant.name from null to "Giant"
20 create cast type Operation
21 set cast.name from null to "smash"
22 add cast to giant.operations at 0
23 create knight type Class
24 set knight.name from null to "Knight"
25 create smash type Operation
26 set smash.name from null to "smash"
27 add smash to knight.operations at 0
28 create mage type Class
29 set mage.name from null to "Mage"
30 session "Bob-01"
31 create leftGen type Generalization
32 set leftGen.general from null to character
33 set troll.generalization to leftGen
34 set character.name from "Character" to "Hero"
35 unset troll.generalization from leftGen to null
       composite l1
36 set knight.generalization to leftGen composite l1
37 move target in attack.parameters from 1 to 2
38 unset cast.name from "cast" to null composite l2
39 remove cast from giant.operations at 0 composite l2
40 delete cast composite l2
41 unset giant.name from "Giant" to null composite l2
42 delete giant composite l2
43 set troll.name from "Troll" to "Ogre"
```

Such a representation is particularly suitable to identify recent changes of the model since the last version. For example, if we can identify that changes recorded for the previous version is up to right before the editing session Bob-01 (lines 1-29) of the model, it can readily identify the changes that have been made to the model since then (i.e. in session Bob-01 – lines 30-43) instead of having to rediscover them through expensive state-based model differencing.

For the sake of readability, the format of change-based persistence presented in Listing 7 is a simplified version. The real format is in XML-like-format. For example, change event session "Jane-01" is persisted as:

<session id="Jane-01" time="20190923181841687GMT"/>

and set character.name from null to "Character" is persisted as:

<set-eattribute eclass="Class" name="name" target = "character"> <old-value literal=null/> <value literal = "Character"/> </set-eattribute>.

3.1 Prototype Implementation

This work has implemented a prototype [12] of the change-based model persistence format – the prototype is named EMF CBP – using the notification facilities provided by the Eclipse Modelling Framework. In the implementation, the prototype uses a subclass of EMF's EContentAdapter (ChangeEventAdapter) to receive and record Notification events produced by the framework for every model-element level change.

Since not all change events are relevant to change-based persistence (e.g. EMF also produces change notifications

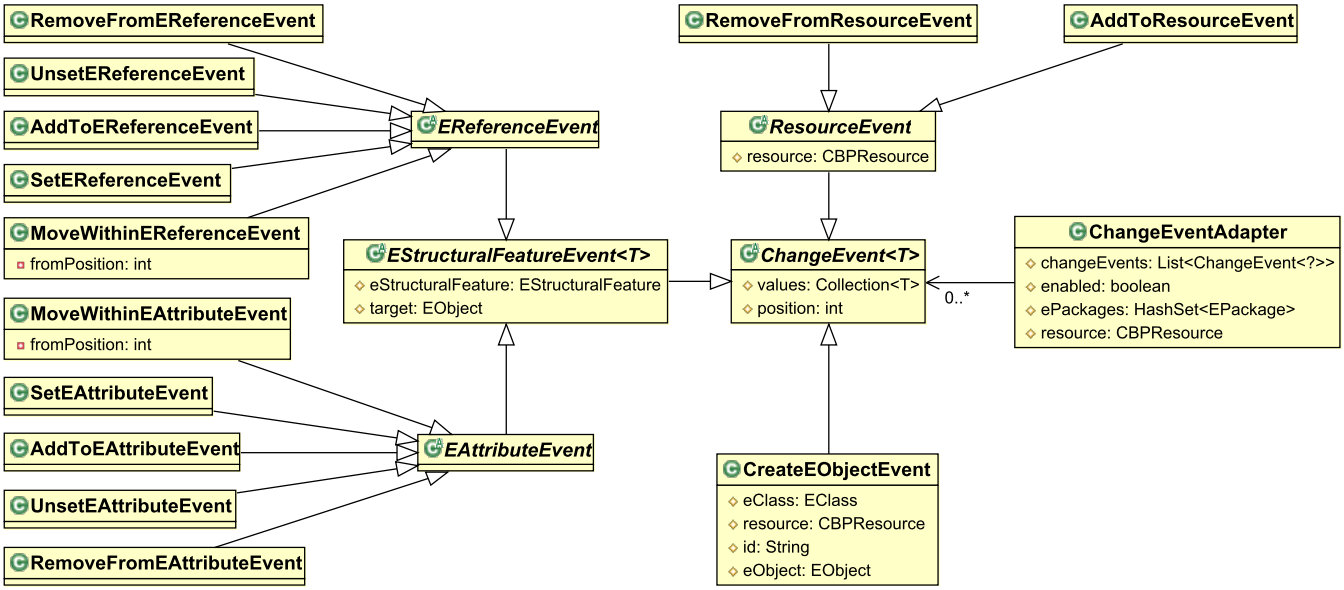


Fig. 2: Event classes to represent changes of models.

when listeners/adapted are added/removed from the model), this work has defined a set of event classes to represent events of interest. The event classes are depicted in Figure 2 as subclasses of the ChangeEvent abstract class.

The ChangeEvent class has a multi-valued values attribute which can accommodate both single-valued (e.g. set/add) or multi-valued events (e.g. addAll/removeAll). ChangeEvent can also accommodate different types of values, such as EObjects for EReferenceEvents, and primitive values (e.g. Integer, String) for EAttributeEvents. The ChangeEvent class also has a position attribute to hold the index of an EObject or a literal when they are added to a Resource, EReference, or EAttribute with multiple values.

Every time an EObject is added to the model, a CreateEObjectEvent and an AddToResourceEvent are recorded. When an EObject is deleted, or moved to a containment EReference deeper in the model, a RemoveFromResourceEvent is recorded.

Listing 8: Simplified Java code to handle notification events.

```
1  public class ChangeEventAdapter extends
       EContentAdapter {
2    ...
3    @override
4    public void notifyChanged(Notification n) {
5      ...
6      switch (n.getEventType()) {
7        ... // other events
8        case Notification.UNSET: {
9        if (n.getNotifier() instanceof EObject) {
10         EStructuralFeature feature = (
               EStructuralFeature) n.getFeature();
11         if (feature instanceof EAttribute) {
12         event = new UnsetEAttributeEvent();
13       } else if (feature instanceof EReference) {
14         event = new UnsetEReferenceEvent();
15       }
16       } break;
17     }
18   ... // other events
```

The ChangeEventAdapter receives EMF change notifications in its notifyChanged() method and filters and transforms them into appropriate change events. As an example of how notifications are filtered and transformed, Listing 8 shows how the prototype handles Notification.UNSET events based on the type of the changed feature i.e. an UnsetEAttributeEvent is instantiated if the feature of the notifier is an EAttribute, or an UnsetEReferenceEvent is created if the notifier is an EReference. The transformed instances are then stored into a list of events in ChangeEventAdapter (changeEvents) for persistence.

To integrate seamlessly with the EMF framework and to eventually support multiple concrete change-based serialisation formats (e.g. XML-formatted representation for readability and binary for performance/size), the prototype implemented CBPResource abstract class, that extends EMF's built-in ResourceImpl class. The role of the abstract class is to encapsulate all change recording functionality while the role of its concrete subclasses is to implement serialisation and de-serialisation. For example, CBPXMLResourceImpl persists changes in a line-based format where every change is serialised as a single-line XML document. In this way, when a model changes, the prototype can append the new changes to the end of the model file without needing to serialise the entire model again. The prototype has also implemented a CBPXMLResourceFactory class that extends EMF's ResourceFactoryImpl, as the factory class for change-based models. Figure 3 shows the relationships between these classes.

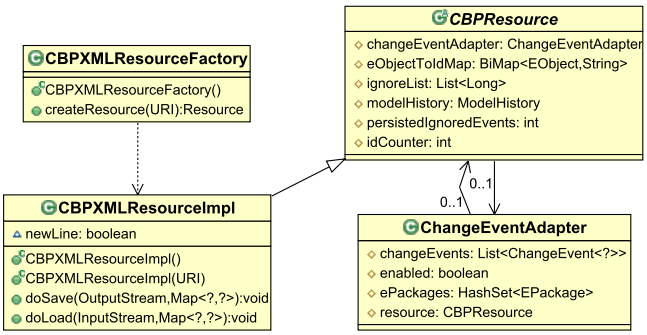


Fig. 3: Factory, resources, and ChangeEventAdapter classes.

Listing 9 shows an example how to use the prototype in a Java code. Lines 1-8 demonstrate how to initialise and save a model using the prototype. First, the code creates an instance of CBPResource, cbpResource, using CBPXMLResourceFactory and specify its file as helloworld.cbpxml using URI. The code than executes method startNewSession of cbpResource. This method adds a change event to indicate the start of editing session as depicted at line 1 in Listings 4 and **??**. The code then uses UMLFactory to create an element, model, of UML2's Model. The code adds element model into cbpResource and set the name to "Hello World". The code then saves the model in change-based format using method save and then unload cbpResource. Lines 9-12 demonstrate how to replay (load) the model that previously has been saved and then print the name of the first element in cbpResource which is expected to print "Hello World" text.

Listing 9: An example how to use CBPResource in Java code.

```
1  /* initialise, save, and unload */
2  CBPResource cbpResource = (CBPResource) (new
       CBPXMLResourceFactory()).createResource(URI.
       createFileURI("helloworld.cbpxml"));
3  cbpResource.startNewSession("Initial");
4  Model model = UMLFactory.eINSTANCE.createModel();
5  cbpResource.getContents().add(model);
6  model.setName("Hello World");
7  cbpResource.save(null);
8  cbpResource.unload();
9
10 /* load and print */
11 cbpResource.load(null);
12 model = (Model) cbpResource.getContents().get(0);
13 System.out.println(model.getName()); // expected
       output: "Hello World"
```

3.2 Benefits and Novel Capabilities

The proposed change-based model persistence also has the potential to deliver a wide range of benefits and novel capabilities, compared to the currently prevalent state-based representations, some of which are discussed below.

- With appropriate tool support, modellers will be able to "replay" (part of) the change history of a model (e.g. to understand design decisions made by other developers, for training purposes). In state-based approaches, this can be partly achieved if models are stored in a version-control repository (e.g. Git). However, the granularity would only be at the commit level.
- By analysing models serialised in the proposed representation, modelling language and tool vendors will be able to develop deeper insights into how modellers actually use these languages/tools in practice and utilise this information to guide the evolution of the language/tool. An early work to deliver this benefit can be found in [13].
- By attaching additional information to each session (e.g. the id of the developer, references to external documents/URLs), sequences of changes can be traced back to the developer that made them, or to requirements/bug reports that triggered them.
- Persisting changes to large models after an editing session will be significantly faster compared to serialising the entire state of the model, as only changes made during the session will need to be appended to the model file. The evaluation of this benefit can be found in [14, 15].
- The performance and precision of model comparison and merging can be substantially improved, particularly for large models with shared editing histories. This paper aims to deliver this benefit.

3.3 Challenges

Change-based model persistence also comes with a number of challenges, such as loading overhead and fast-growing model files. While, as discussed above, persisting changes to large models is expected to be much faster and resource-efficient compared to state-based approaches, loading models into memory by naively replaying the entire change history is expected to have a significant overhead. To address this challenge, we have developed two solutions that reduces the cost of change-based model loading, firstly, by recording and ignoring events – events that are later overridden or cancelled out by other events – [14] and, secondly, by proposing a hybrid model persistence format which models are loaded from stated-based persistence but changes are persisted into both change-based and state-based representation [15].

For the fast-growing model files challenge, persisting models in a change-based format means that model files will keep growing in size during their evolution significantly faster than their state-based counterparts. To address this challenge, one can proposes sound change-compression operations (e.g. remove older/unused information) that can be used to reduce the size of a model in a controlled

way or develops a compact textual format that will minimise the amount of space required to record a change (a textual line-separated format is desirable to maintain compatibility with file-based version control systems).

4 Existing Conflict Detections in Models

4.1 State-based Conflict Detection (EMF Compare)

In state-based model comparison, a conflict occurs when the states of an element or a feature are different in two different versions of a model that are being compared. In other words, it can be said that the applied change events that cause the difference are in conflict since they produce two different states – in a previous shared version, these states are equal. State-based persistence does not record change events that cause the differences, thus the change events has to be derived through model differencing [2,9].

Let's say that we have three versions of a model, the original shared version m_o and two other modified versions: the left version m_l and the right version m_r. There are also two sets of derived operations, C_L and C_R. These sets are obtained by differencing m_l to m_o and m_r to M_O using an LCS algorithm as explained in Section **??**, where $C_L = \{c_{l1}, c_{l2}, ..., c_{olg}\}$, $C_R = \{c_{r1}, c_{r2}, ..., c_{rh}\}$, $g = |C_L|$, and $h = |C_R|$. Applying C_L to model m_o transforms it into model m_l, and applying C_R to model m_o transforms it into model m_r. These derived change events are be used to detect conflicts using (1), (3), and (2).

If the method in Section **??** is used – as executed in EMF Compare – to derive C_L from the left and original versions – Bob's and Jane's versions – in Figure 1 and present it in the format of change events, the following list are obtained.

Listing 10: The derived change events made by Bob to produce the right model in Figure 1b (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 composite l1
set leftGen.general from null to character composite
    l1
set knight.generalization from null to leftGen
    composite l1
unset cast.name from "cast" to null composite l2
remove cast from giant.operations at 0 composite l2
delete cast composite l2
unset giant.name from "Giant" to null composite l3
remove giant from resource at 2 composite l3
delete giant composite l3
```

And following list is the derived change events for C_R that are obtained from the right and original versions – Alice's and Jane's versions – in Figure 1.

Listing 11: The derived change events (operations) made by Alice to produce the right model in Figure 1c (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 composite r1
add smash to giant.operations at 0 composite r1
create rightGen type Generalization composite r2
set rightGen.general to character composite r2
set mage.generalization to rightGen composite r2
remove cast from giant.operations at composite r3
add cast to mage.operations at 0 composite r3
```

Real Conflict. In state-based model comparison, two change events, c_l and c_r, are in conflict if both are applied to a same element e_o but produce two different eventual states where ! is used as the operator for expressing that two change events are in conflict (1). EMF Compare [2] classifies this conflict as a REAL conflict. For example, Bob changed the name of troll to "Ogre" (Listing 10) while Alice modified it to "Orc" (Listing 11).

$$e_o + c_l \not\equiv e_o + c_r \Rightarrow c_l \,!\, c_r \tag{1}$$

Non-applicability. A REAL conflict also occurs when applying a change event c_l to element e_o makes c_r inapplicable to element e_o. Therefore, change events c_l and c_r are in conflict (2). For instance, Alice moved operation smash from class Knight to class Giant (Listing 11) but this class was deleted by Bob (Listing 10). Deleting class Giant makes the move inapplicable.

$$(e_o + c_r \not\equiv e_o) \wedge (e_o + c_l + c_r \equiv e_o + c_l) \Rightarrow c_l \,!\, c_r \tag{2}$$

Pseudo Conflict. A conflict is classified as PSEUDO if the eventual states produced are equivalent. The PSEUDO means the conflict can be automatically resolved by choosing any of the conflicting changes since any of the changes produces the same eventual states (3). Symbol $!_p$ is used as the operator for expressing that two change events are in PSEUDO conflict. For example, both Bob and Alice changed the name of element character from "Character" to "Hero" (Listings 10 and 11).

$$e_o + c_l \equiv e_o + c_r \Rightarrow c_l \,!_p\, c_r \tag{3}$$

Using (1), (2), and (3) and information in Listings 10 and 11, four conflicts can be identified – presented in Table 1 along with their conflicting change events. Conflict EC1 is a pseudo conflict since both modify the same class character's feature name resulting the same end states, "Hero" or "Hero". Conflict EC2 is a REAL 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 REAL non-applicability conflicts since if operation cast is deleted first then it cannot be moved – removed and added – from class giant's operations to class mage's operations, and if class giant is deleted first then operation smash cannot be moved – removed and added – from class knight's operations to class giant's operations.

Conflict detection in state-based comparison might not be accurate since the derived differences/change events might not reflect the real historical changes of a model. For example, EMF Compare [2], a tool that perform

Table 1: Conflicting change events identified using EMF Compare based on the case in Figure 1.

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"	real
EC2	set troll.name from "Troll" to "Ogre"	set troll.name from "Troll" to "Orc	real
EC3	delete cast	remove cast from giant.operations at 0 add cast to mage.operations at 0	real, dependency
EC4	delete smash	remove smash from knight.operations at 0 add smash to giant.operations at 0	real, dependency

state-based model comparison, does not detect that Alice and Bob modified the same element – parameter target – as indicated by line 29 in List. 6 and line 35 in List. 5. Using an LCS algorithm, the derived change events 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 (1), both change events are not in conflict since both change events modify two different elements, target and gem. The result is different if change-based approach is employed to detect conflicts using the change event records in Listings 5 and 6 which is explained in Section 4.2.

4.2 Change-based Conflict Detection (EMF Store)

EMF Store [17] is a product that implements change-based model persistence for EMF models. It is a collaborative repository and versioning system that is specifically designed for models to answer existing versioning systems, such as Git and SVN, that focus heavily on text-based files [10]. EMF Store follows the following rules to identify conflict between change events [16].

Non-commutability. In EMF Store, change events c_l and c_r are in conflict if applying them in different order to a same element e_o produces two different eventual states. For example, Alice changed the name of class Troll to "Orc" (Listing 6) while Bob renamed it to "Ogre" (Listing 5). Applying Alice's change first to Bob's change results in the class' name equals to "Ogre", or "Orc" if the order is reversed.

$$e_o + c_l + c_r \not\equiv e_o + c_l + c_r \Rightarrow c_l \,!\, c_r \tag{4}$$

However, after examining the implementation [18], even though two different change events produce equivalent eventual states, both change events are still treated in conflict by EMF Store. For example, both Bob and Alice changed the name of element character from "Character" to "Hero" (Listing 5 line 34 and Listing 6 line 39). The reason is if we apply Bob's set event first, it changes character's name from "Character" to "Hero". It is important to notice that after applying Bob's set event, the eventual value of character's name is "Hero". Applying Alice's set event with previous value "Character" is inapplicable since it makes the sequence of the change events inconsistent; Bob's set event produces end value "Hero" which is not the previous value changed by Alice's set event, which is "Character". The same inconsistency still occurs even we apply these set events in different order.

Co-modifiability. Thus, this leads to a new definition that conflict occurs when two different change events modify a same element or feature regardless the equivalency of the eventual states that they produce.

$$(e_o + c_l \equiv e_o + c_r) \vee (e_o + c_l \not\equiv e_o + c_r) \Rightarrow c_l \,!\, c_r \tag{5}$$

Non-applicability. This non-applicability rule is the same with the non-applicability rule in the state-based conflict detection. Essentially, a conflict occurs when applying a change event c_l to element e_o makes c_r inapplicable to element e_o. For instance, Alice moved operation smash from class Knight to class Giant (Listing 6) but this class was deleted by Bob (Listing 5). Deleting class Giant makes the move inapplicable.

$$(e_o + c_r \not\equiv e_o) \wedge (e_o + c_l + c_r \equiv e_o + c_l) \Rightarrow c_l \,!\, _r \tag{6}$$

Composite. If change event c_l is in conflict with change event c_r where c_r is a member of a set of composite change event cc_r then change event c_l is also in conflict with each change event c_n in composite change event cc_r. For example, the deletion of class Giant is part of composite event l2 (Listing 5) and the addition of operation smash to class Giant is part of composite event r1 (Listing 6). Since they are in conflict according to (6), all other change events in their composite events, l2 and r1, are also in conflict.

$$c_l \,!\, c_r \wedge c_r \in cc_r \Rightarrow c_l \,!\, \forall c_{rn} | c_{rn} \in cc_r \tag{7}$$

In change-based conflict detection, all change events applied to a model are already available in change-based persistence, thus the change events do not need to be derived trough a diffing process. The availability of real historical changes can improve the accuracy of change detection since elements that have been changed can be identified according to fact – not derivation. In consequence, it can detect conflicts that cannot be detected by state-based conflict detection. For example, in Listing 6 line 31, parameter target has been moved from index 1 to 0, while in Listing 5 line 37, it was moved from index 1 to 2. Since both change events modified the same parameter target, both change events can be identified in conflict

Table 2: Conflicting change events identified using EMF Store in Listings 6 and 5.

ID	Left Change Events (Bob)	Right Change Events (Alice)	Type
ES1	set troll.generalization from null to left Gen unset troll.generalization from leftGen to null set knight.generalization from null to leftGen	set troll.generalization from null to rightGen unset troll.generalization from rightGen to null set mage.generalization from null to rightGen	co-modifiability composite
ES2	set character.name from "Character" to"Hero"	set character.name from "Character" to "Hero"	co-modifiability
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 remove smash from knight.operations at 0 add smash to giant.operations at 1	non-applicability composite
ES5	set troll.name from "Troll" to "Ogre"	set troll.name from "Troll" to "Orc"	co-modifiability

using (5); the same parameter target is modified by two different change events.

The drawback of EMF Store is that it treats two change events that modify a same element as they are in conflict regardless of the end states that they produce to the element [19]. In common sense, two changes should not be in conflict if they are applied to a same element or feature and produce same eventual states. Moreover, EMF Store it does not have a classification that separates conflicts into REAL or PSEUDO conflicts, such as in EMF Compare, to automate conflict resolution. For example, two change events in Listing 6 at line 39 and Listing 5 at line 35, that change the same feature name from "Character" to the same value "Hero", are treated in conflict (Table 2 id ES2) using (5). EMF Compare classifies this kind of conflict as PSEUDO conflict which can be automatically resolved in merging only by selecting one of the conflicting change events and ignoring the other one.

Excluding eventual states in detecting conflicts also causes all change events related to troll's generalization to be in conflict; all the feature's left-side events are in conflict with all its right-side events (Table 2, ES1). Using the co-modifiability (5) rule, we can detect that the setting and unsetting of troll' generalization to leftGen and null (Listing 5 lines 33, 35) are in conflict with the setting and unsetting of troll' generalization to rightGen and null (Listing 6 lines 38, 40). Moreoever, using composite (7) rule, we can also identify that the setting of knight' generalization to leftGen (Listing 5 line 36) and the setting of mage' generalization to rightGen (Listing 6 line 41) are also part of the conflict ES1 since both events are in the same composite move events, l1 and r3, with the unsetting of troll' generalization to null (Listing 5 line 35, Listing 6 line 38).

In state-based conflict detection, 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 different *derived* change events that modify class troll's feature generalization in parallel.

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 can be detected with the co-modifiability (5) rule. The states of troll's name have been simultaneously modified to "Ogre" or "Orc".

The summary of the advantages and drawbacks between EMF Compare and EMF Store in detecting conflicts are presented in Table 3. The state-based approach, represented by EMF Compare [2], does come with drawbacks. First, it cannot detect conflicts as accurate as change-based approach can as their changes are derived – not the real historical changes. Second, EMF Compare uses 3-way model comparison [2] thus hypothetically its conflict detection should perform relatively slower than the change-based approach, since it has to perform state-based model differencing twice to derive changes events: changes events between left and original versions, and change events between right and original versions.

Change-based model conflict detection [16], represented by EMF Store [10], also has drawbacks. EMF Store purely works on comparing change events and detects conflicts based on predefined rules; it does not consider the eventual states of two versions that are being compared. Thus, two change events that modifies a same feature are considered in conflict even though both change events produce the same eventual states. This condition can lead EMF Store to oversensitive conflict detection. In terms of performance, as has been investigated in [9], the change-based approach is faster than its state-based counterparts in model differencing. Thus, it is expected that it can also performs better that the state-based approach in detecting conflicts.

Table 3: The advantages and drawbacks of EMF Compare and EMF Store in detecting conflicts.

Dimension	State-based Conflict Detection (EMF Compare)	Change-based Conflict Detection (EMF Store)
Advantages	- detect PSEUDO conflict which can be automatically resolved when merging - conflicts detected are optimal since changes are derived thus avoid oversensitive conflict detection	- more accurate in detecting conflicts since changes are real history
Drawbacks	- less accurate in detecting conflicts since changes are derived – not real changes - in large models, its performance should be slower the change-based approach since it performs 3-way comparison which requires 2-times model differencing to derive changes	- treat all conflicts as REAL conflicts which demand user intervention for resolution - can be oversensitive in detecting conflicts since eventual states are not considered - in large models with moderate changes, it should perform faster than the state-based approach – no need to derive changes since they are available already

5 EMF CBP Conflict Detection

This work proposes a change-based model conflict detection that also considers the eventual states of modified elements. Thus, the performance and accuracy of model conflict detection can be improved, compared to state-based approach, without being oversensitive.

Compared to the state-based model differencing of EMF Compare, the change-based model proposed in this work consists of three phases: event loading, element tree construction, and diff computation. The comparison is not performed over all the elements of the model as opposed to state-based model differencing; instead, the approach only needs to compare the last set of changes from the source and reference model. The last set of changes can be identified easily by finding their last common change. A simplified class diagram of the approach's implementation [12] is depicted in Figure 4. The three phases are described in detail in the following Sections.

The model conflict detection proposed in this study basically performs similar procedure to the phases of change-based model differencing discussed in Chapter **??** except that (1) it only executes event loading and element tree construction phases and replaces diff computation phase with conflict computation phase, and (2) during element tree construction, it maps change events to the elements, features, and values that they modify. The change event mapping and conflict computation are discussed in the following Sections.

5.1 Event Loading

In the event loading phase, the implementation loads change events recorded in two change-based model persistence 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 1-29 in Listing 4 are skipped. Only lines starting from line 30 in Listings 5 and 6 are loaded, yielding two partial – left and right – change-event models.

5.2 Element Tree

An element tree is a representation of the changes of model elements in the source and reference models. It contains detailed information about elements and their properties. It contains similar information to that captured in change lists in state-based model persistence, 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.

5.2.1 Left Side In the first change event in Listing 5 at line 30, the change event is a session event. It marks that the all the following change events until the final line or next session event are persisted in one batch when saving. At line 31, we can identify that Bob created a Generalization with id leftGen. Thus, in elementTree, an element with id leftGen is also created. To mark that an element is newly created in the session, we put a '+' sign at the left lower box of element leftGen in Figure 5.

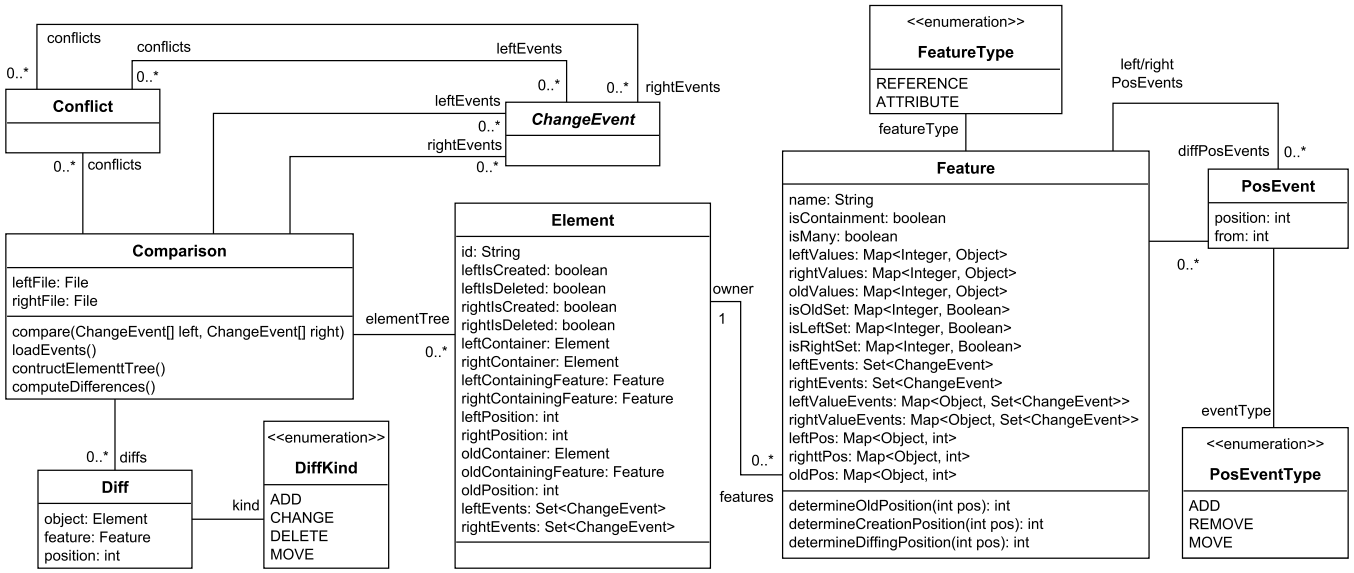


Fig. 4: A class diagram showing the core components of the change-based approach to speed up model differencing and conflict detection.

At line 32, the feature general of leftGen is set to character. From the change event, we can recognise that character has been existed since the previous version since it has never been created in the current editing session. Thus, we create an element with id character as well as the feature general of leftGen and put them in elementTree and then set the value of general to character on the left side. We also do the same routine to troll and generalization at line 33, adding element troll and feature generalization to elementTree and set the value of feature generalization to leftGen on the left side of the elementTree.

Change event at line 34, change character's name from "Character" to "Hero". From the change event, we can identify that character has been existed before. Thus, we create element character and feature name into elementTree. We also set the value of name to "Hero" on the left side. Since this set change event is the first event for character's name, we can infer that original value of name is "Character". Thus, we set name's value to "Character" on the original side. The value of name on the right side is also set to "Character" but will be modified later once we process the right change events (Alice's change events) if there is any change event that affects it. The same routine is also applied when we process the change event at line 43 later.

Lines 35 and 36 are the changes events of composite move event l1. Element leftGen is removed (unset) from troll's generalization and then is assigned (set) to knight's generalization. From these change events, we can identify that element knight also already existed since the original version. Thus, we add it into elementTree together with its generalization feature. Element troll and its generalization feature are not added into elementTree any more since they were already added when processing line 33. In elementTree, we set troll's generalization to null since element leftGen is moved to knight's generalization.

At line 37, target is moved from index 1 to 2 in attack's parameters. From the change event, we can identify that there has been element target contained in attack's parameters at index 1 since the original version. Thus, we put element target and element attack and its parameters feature into elementTree. We also create a map on the left side with a key '2' and a value that points to element target for feature parameters, indicating target is at index 2 in the left version. Since it is the first change event that moves target, we can decide that target is at index 1 in the original version. Thus, we create another map on the original side a map on the left side with a key '1' and a value that also points to target. We also perform this routine to the right side of feature parameters, creating a map with a key '1' and a value that also points to target. It will be modified later once we process the right change events (Alice's change events) if there is any change event that affects the index of target.

Lines 38 to 42 are the change events of composite delete event l2; a deletion of element giant. A deletion of an element unsets all the features of the elements and its sub elements, removes the sub elements from their containers, and deletes the sub elements and the element from the model. As can be noticed, at line 38, the value of cast's name is unset from "cast" to null. From the change event, we know that cast has been existed since the original version. Thus, we add element cast and its feature name to elementTree and set its value null on the left side and "cast" on the origin and right sides.

At line 39, cast is removed from giant's operations at index 0. From it, we can identify that giant and its feature

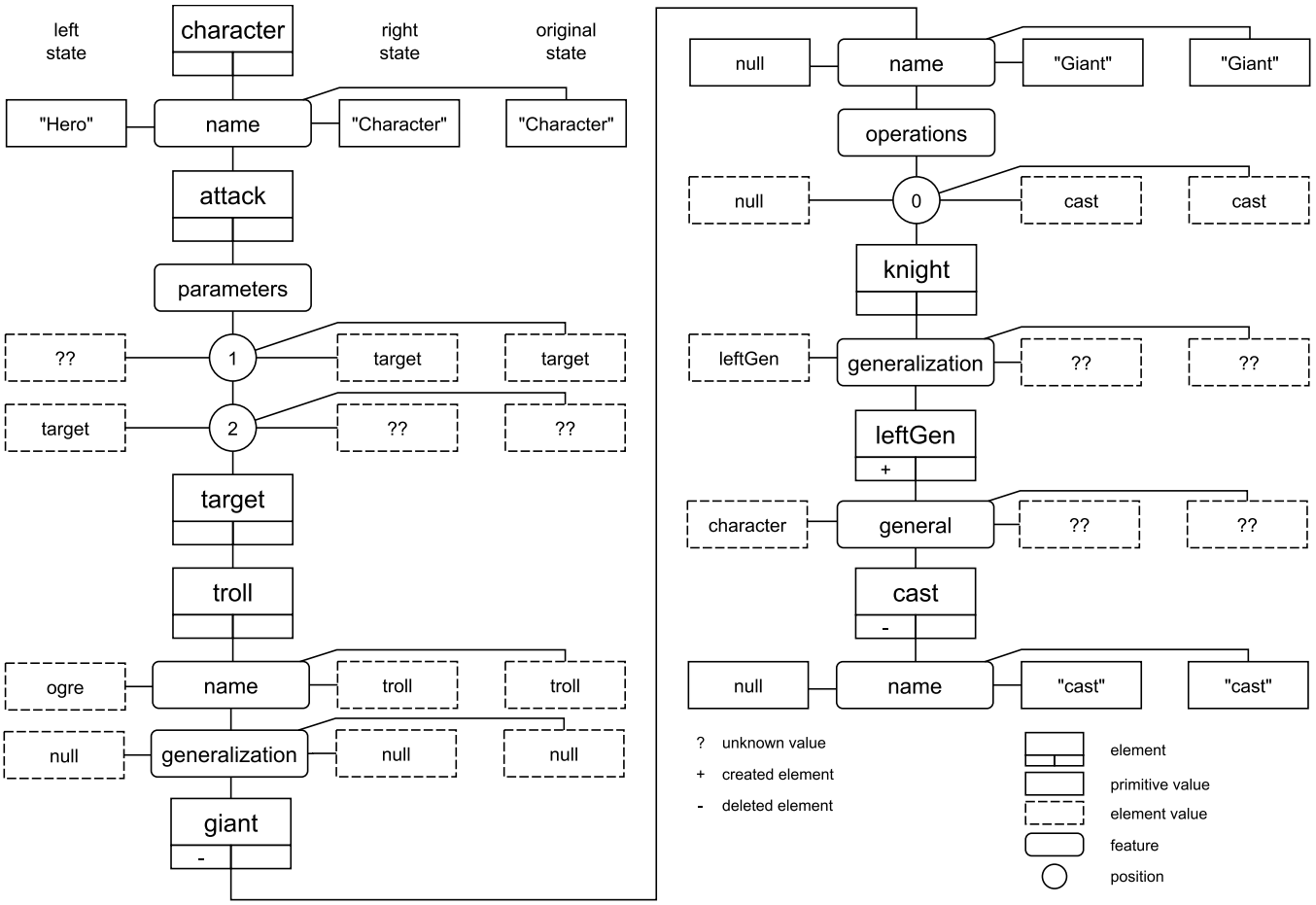


Fig. 5: An element tree constructed using information contained in CBPs in Listing 5 (all left change events only).

operations exists, and cast is contained in giant's operations at index 0 in the original version. Thus, we create element giant and its feature operations in elementTree. Three maps also are created in operations for the three sides. Each map contains a key '0', indicating index, and a value that points to element cast except on the left side the value is null since cast is removed from giant's operations. The deletion of cast at line 40 marks cast in elementTree with '-' sign on the left side indicating that the element is deleted from the model in the left version.

Change event at line 41 is similar to change event at line At line 38 except that it is applied to giant's name. Since giant has existed in elementTree, only the feature name is added. Its value is set to null on the left side and "Giant" on the origin and right sides. The deletion of giant at line 42 marks giant in elementTree with '-' sign indicating that the element is deleted from the model in the left version.

Figure 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.

5.2.2 Right Side In Listing 6, similar to processing the left change events, the processing of the right change events (Alice's version) starts with processing the session event at line 30. At line 31, target is moved from index 1 to 0 in attack's parameters. Since the index of target is already determined when processing the change event, we only determine the index of target on the right side. We unset the value of key '1' on the right side to null and create a new key '0' that maps its value to target.

Composite move event r1 at lines 32 and 33 moves smash from knight's operations to giant's operations. From this move event, we can identify smash is no longer in knight's operations but contained in giant's operations on the right side. Element smash has never existed in elementTree. So, we create and add smash to knight's operations at index 0 on the origin side and to giant's operations at index 0 on the right side. Since smash is not modified on the left side and no other change events applied to knight's operations, we can determine that smash is at index 0 in giant's operations on the left side.

Lines 34 to 35 are change events that constitute composite move event r2. It moves cast from giant's operations to mage's operations. From this move event, we can identify

cast is no longer in giant's operations but now exists in mage's operations on the right side. Element mage and its feature operations have never existed in elementTree. So, we create and add them to elementTree and add cast to mage's operations on the right side.

At line 36, we can identify that Alice created a Generalization with id rightGen. Thus, in elementTree, an element with id rightGen is also created. Since it has just been created in the active session, the element is marked with '+' sign in elementTree on the right side. At line 37, we can also identify that feature general should be added to rightGen in elementTree and the value is set to character on the right side. We also set mage's operations to rightGen on the right side of elementTree according to the change event at line 38.

Change event at line 39, change character's name from "Character" to "Hero". Since character and its feature name already exists in elementTree, we only set name's value to "Hero" on the right side; the original value has already been assigned when processing left change events. We apply the same routine when processing the change event at line 42 later.

Composite move event r3 at lines 40 and 41 moves rightGen from troll's generalization to mage's generalization. From this move event, on the right side, we can identify rightGen is no longer in troll's generalization but exists in mage's generalization. Since it's the first mage's generalization is modified, we create and add the feature to mage in elementTree. On the right side of elementTree, we unset troll's generalization to null and assign rightGen to mage's generalization.

Figure 6 exhibits the state of the elementTree after both sides' change events have been processed.

5.2.3 Construction Procedure The construction of elementTree that has been explained follows the steps shown in Figure 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 $I_{OL} \cup I_{RR} \rightarrow S_R$. 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, $I_{OL} \cup I_{OR} \rightarrow S_O$. Finally, we also use the information I_{OR} to update the partial state of the left model (step 6), implying that $I_{LL} \cup I_{OR} \rightarrow S_L$.

Algorithm 1 describes the steps presented in Figure 7 in a generic fashion. It iterates through all of 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 are 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 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., character and giant in Listing 5). This targetElement – an instance of class Element in Figure 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 Figure 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 Figure 4 – represents the target element's feature (e.g., name and operations in Listing 6) 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 the feature does exist (line 5).

The value is the value assigned to the feature in a change event (line 5, Algorithm 1). The value can be the type of Element (e.g., element leftGen line 36 in Listing 5) or primitive (e.g., the string "Hero" at line 34 in the Listing 5). The previousValue represents the previous value of the modified feature (line 6, Algorithm 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 events 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, Algorithm 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 obtain that the index of cast is 0 (line 35 in Listing 6) as the change event type is add. In a remove change event, we can only get the previousIndex of cast, that is 1 (line 35 in Listing 6), 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

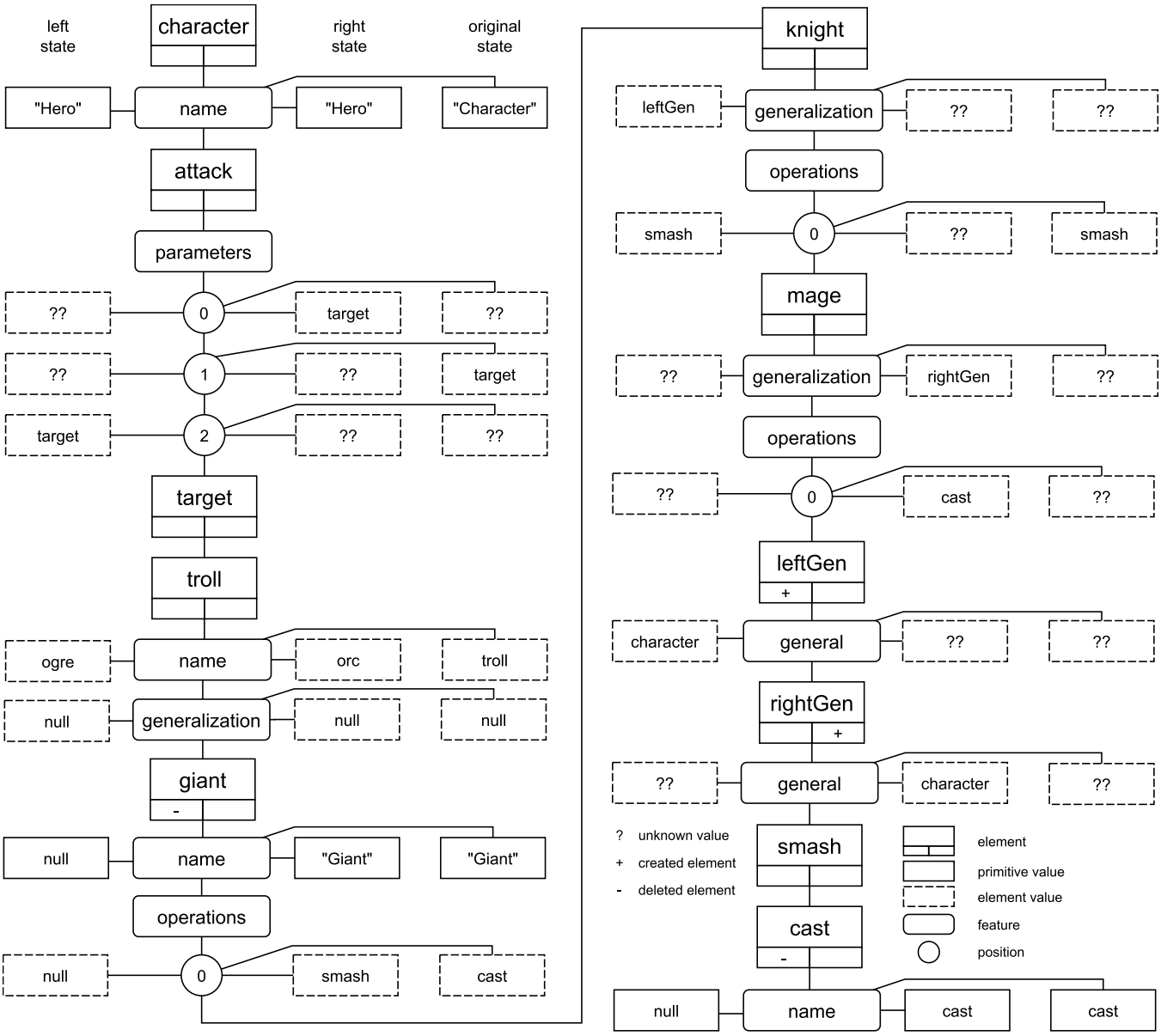


Fig. 6: An element tree constructed using information contained in CBPs in Listings 5 and 6 (all left and right change events).

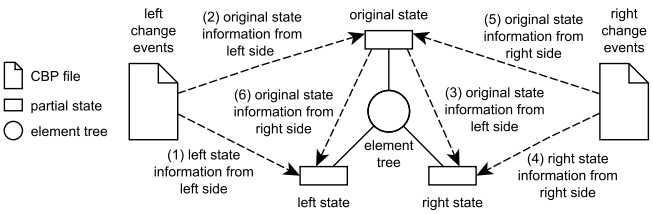


Fig. 7: Steps in Element Tree construction.

is moved from a previous index to a new one (line 31 in Listing 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.

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       oppositeFeatureEventList ← getOppositeFeatureEventList(feature,
           side);
15       oppositeIndex ← calculateOppositeIndex(oppositeFeatureEventList,
           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
```

5.3 Change Event Mapping

Using the information contained in change-based model persistence in Listings 6 and 5, we can construct an element tree as depicted in Figure 6 using the construction method presented in Section 5.2. During the the construction, change events in Listings 6 and 5 are mapped to the affected elements, features, and values which act as the keys of the mapping. The relationships are stored in attributes leftEvents and rightEvents of class Element and leftEvents, rightEvents, leftValueEvents, and rightValueEvents of class Feature in Figure 4. This registration forms many-to-many relationships between the keys and change events. In detail, the keys are element for elements, or a combination of element-feature for single-valued features or 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 6 and 5 is in Table 4. The application of this mapping is presented in Section 5.5.1.

Table 4: The mapping of elements, features, and values in Figure 6 to the events that affect them.

Key	Left Events	Right Events
character	cl32, cl34	cr37, cr39
character.name	cl34	cr39
attack	cl37	cr31
attack.parameters.target	cl37	cr31
target	cl37	cr31
trcll	cl33, cl35	cr38, cr40
trcll.name	cl43	cr42
trcll.generalization	cl33, cl35	cr38, cr40
giant	cl39, cl40, cl41, cl42	cr33, cr34
giant.name	cl40	
giant.operations.cast	cl39	cr34
giant.operations.smash		cr33
knight	cl36	cr32
knight.generalization	cl36	
knight.operations.smash		cr32
mage		cr35, cr41
mage.generalization		cr41
mage.operations.cast		cr35
leftGen	cl31, cl32, cl33, cl35, cl36	
leftGen.general	cl32	
rightGen		cr36, cr37, cr38, cr40, cr41
rightGen.general		cr37
smash		cr32, cr33
cast	cl38, cl39, cl40	cr34, cr35
cast.name	cl38	

c: change event; l: left side; r: right side; n: line number in change-based model persistence

5.4 Theoretical Foundation

In the proposed 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 the modified models. Two sequences of change events that produce two eventual states that are equal to an original state are not treated as in conflict. The original and eventual states are already calculated during the construction of the element tree so that we do not to calculate them again in the conflict computation phases. Since all change events are also recorded to every element, feature, and value that they affected, we can retrieve all related change events that produce the eventual state of an element or feature.

Let's say that we have the original state of an element e_o. We also have a set of change events $C_L = \{c_{l1}, c_{l2}, ..., c_{lg}\}$ that we apply to e_o that changes its state to element e_l and $g = |C_L|$.

$$e_o + c_{l1} + c_{l2} + ... + c_{lg} \rightarrow e_l \tag{8}$$

We also have another set of change events $C_R = \{c_{r1}, c_{r2}, ..., c_{rh}\}$ that we apply to e_o that produces element e_r and $h = |C_R|$.

$$e_o + c_{r1} + c_{r2} + ... + c_{rh} \rightarrow e_r \tag{9}$$

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

$$\begin{aligned}&(e_o \equiv e_l \wedge e_o \equiv e_r) \vee |C_L| = 0 \vee |C_R| = 0 \Rightarrow \\ &\neg(\forall c_l \,!\, \forall c_r) \mid c_l \in C_L, c_r \in C_R\end{aligned} \tag{10}$$

Conflict. A conflict occurs when one or both states, e_l or/and e_r, are not equivalent to the original state e_o, and, at least, there is a change event applied on each side of the element; we can conclude that change event set C_L is in conflict with the change event set C_R.

$$\begin{aligned}&(e_o \not\equiv e_l \vee e_o \not\equiv e_r) \wedge (|C_L| > 0 \wedge |C_R| > 0) \Rightarrow \\ &\forall c_l \,!\, \forall c_r \mid c_l \in C_L, c_r \in C_R\end{aligned} \tag{11}$$

Pseudo-conflict. As in EMF Compare, we also implement pseudo conflict. Pseudo conflict is a conflict where e_l and e_r are equivalent or one of them is equivalent to e_o thus they can be automatically resolved in conflict resolution without user intervention.

$$\begin{aligned}&(e_o \equiv e_l \vee e_o \equiv e_r \vee e_l \equiv e_r) \wedge (|C_L| > 0 \wedge |C_R| > 0) \\ &\Rightarrow \forall c_l \,!_p \, \forall c_r \mid c_l \in C_L, c_r \in C_R\end{aligned} \tag{12}$$

Figure 8 are some examples how conflict and non-conflict change events are detected in the proposed approach (dashed arrow = left change event, solid arrow = right change events, circle = state). Figure 8a shows the initial state of an element is 'a'. In the figure, the element has not been modified. Thus, no conflict is detected according to (10). In Figure 8b, the element is modified on the right side (version) only. Thus, using (10), no conflict is detected. In the figure, the state of the element is altered from 'a' to 'b' by change event $cr1$, and then altered again to 'c' by change event $cr2$. In Figure 8c, even though an element has been modified on both sides, using (10), no conflict is detected, since both left and right states are

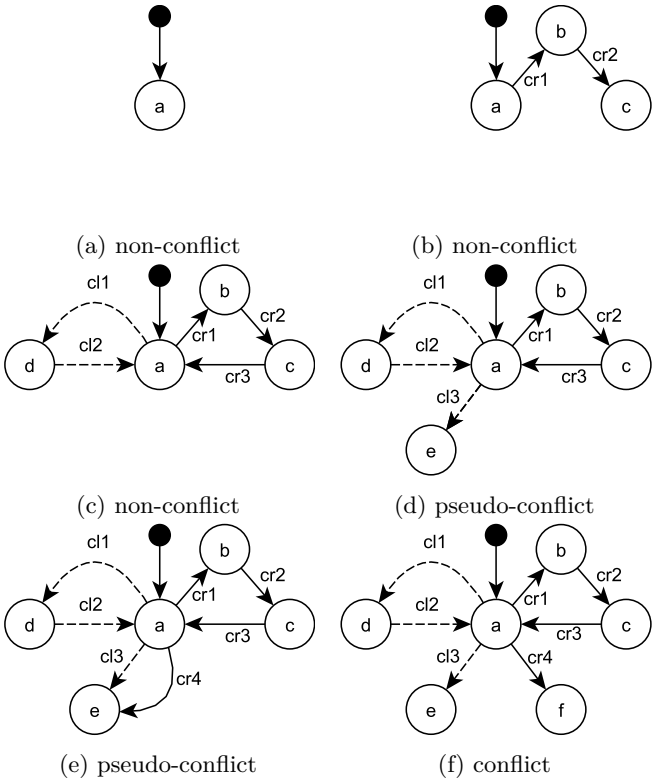


Fig. 8: Conflicting and non-conflicting change events (dashed arrow = left change event, solid arrow = right change events, circle = state).

equal to the original state after the modification. In the figure, both C_L and C_R produces eventual states that are equal to the original state, 'a'.

Using (12), condition in Figure 8d can be detected as a PSEUDO conflict. PSEUDO conflict means that a conflict can be automatically resolved. This means that we can automatically select one of the two conflicting change event sets as the applied change events without needing for human intervention. Since C_R produces the eventual state that is equal to the original state, that is 'a'; it does not have any effect – the changes are not intended or cancelled. Thus, all its change events can be automatically negated. In other words, only change events in C_L that are accepted to produce the eventual state, which is 'e'. Also using (12), condition is 8e can also be detected as another PSEUDO conflict. Both change event sets, C_L and C_R, produce the same eventual state, 'e', that is different from the original state, 'a'. This condition ca be automatically resolved since selecting one of the sets produces the same outcome. With (11), the condition in Figure 8f can be detected as a REAL conflict since change event sets, C_L and C_R, produces two different eventual states. The conflict cannot be automatically resolved and requires user intervention to choose which one is the desired eventual state, 'e' or 'f', and then

the appropriate change event set should be selected to produce the eventual state.

5.5 Conflict Computation

We perform procedure in Algorithm 2 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 (Figure 6) and checking the equivalency of their original and eventual states as well the numbers of change events 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 at lines 2. The algorithm then iterates through all the elements, features, and values in the element three.

5.5.1 Conflict with Deletion At lines 4 to 11 in Algorithm 2, 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 should be in conflict. To get all the related events, the algorithm use functions getAllRelatedLeftEvents(*element*) and getAllRelatedLeftEvents(*element*) (the element acts as a map key to access the change 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.

As an example, when the iteration reach element giant in Figure 6, the algorithm identifies that the element has been deleted only on the left side. Using the map in Table 4, the algorithm then collects all the change events from both sides related to the element giant and its sub-elements. For key giant, it collects change events at lines 39 to 42 for the left side and change events at lines 33 to 34 for the right side. For key giant.name, only left-side change event at line 40 is collected. For key giant.operations.cast, it collects left-side change event at line 39 and right-side change event at line 34. For key giant.operations.smash, only right-side change event at line 33 is collected. For key cast, it collects change events at lines 38 to 40 for the left side and change events at lines 34 to 35 for the right side. For key giant.name, it only left-side change event at line 38 is collected. The collected change events are merged into one set of change events for each side. So, basically, the left events are all events parts of the composite event that deletes the element. The right events consist of events that move operation

Algorithm 2: Algorithm for conflict detection using element tree.

```
   input  : an instance of ElementTree elementTree
 1 begin
 2     conflictList ← ConflictList();
 3     foreach element in elementTree do
           // Handle conflicts with deletion
              ------------------
 4         if isLeftDeleted(element) or
            isRightDeleted(element) then
 5             leftEvents ←
                getAllRelatedLeftEvents(element);
 6             rightEvents ←
                getAllRelatedRightEvents(element);
 7             if size(leftEvents) > 0 and
                size(rightEvents) > 0 then
 8                 conflict ← createConflict(leftEvents,
                    rightEvents);
 9                 if isLeftDeleted(element) and
                    isRightDeleted(element) then
10                     setPseudo(conflict);
11                 end
12                 addConflict(conflict, conflictList);
13                 continue;
14             end
15         end
           // Handle conflicts with cross-container
              move -----------------
16         if (getOriginalContainer(element) <>
            getLeftContainer(element) or
            getOriginalContainingFeature(element) <>
            getLeftContainingFeature(element)) or
            (getOriginalContainer(element) <>
            getRightContainer(element) or
            getOriginalContainingFeature(element) <>
            getRightContainingFeature(element)) then
17             leftEvents ←
                getAllRelatedLeftEvents(element);
18             rightEvents ←
                getAllRelatedRightEvents(element);
19             if size(leftEvents) > 0 and
                size(rightEvents) > 0 then
20                 conflict ← createConflict(leftEvents,
                    rightEvents);
21                 if getLeftContainer(element) =
                    getRightContainer(element) and
                    getLeftContainingFeature(element) =
                    getRightContainingFeature(element
                    then
22                     setPseudo(conflict);
23                 end
24                 addConflict(conflict, conflictList);
25             end
26         end
27         foreach feature in getFeatures(element) do
               // Handle single-valued feature
                  ------------------
28             handleSingleValuedFeature(element, feature,
                conflictList);
               // Handle multi-valued feature
                  ------------------
29             handleMultiValuedFeature(element, feature,
                conflictList);
30         end
31     end
32     return conflictList;
33 end
```

smash from class knight to class giant and events that move operation cast from class giant to class mage. The algorithm then creates a conflict that consists of these events producing conflict CB3 in Table 5. The conflict is not set PSEUDO since, it is only deleted on the left side.

When the iteration reach element cast – the operation of class giant. The same procedure is also executed. It collects left-side change events at lines 33, 38, 39, 40, 41,

Table 5: Conflicting change events in Listings 5 and 6 identified using the proposed change-based conflict detection. The bold identifiers are the keys where the conflicts detected.

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"	real
CB2	move **target** in **character.parameters** from 1 to 2	move **target** in **character.parameters** from 1 to 0	real
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** type Class	remove smash from knight.operations at 0 add smash to **giant**.operations at 1 remove cast from **giant**.operations at 0 add cast to mage.operations at 0	real, dependency
CB4	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 type Class	remove **cast** from giant.operations at 0 add **cast** to mage.operations at 0	real, dependency
CB5	set **character.name** from "Character" to "Hero"	set **character.name** from "Character" to "Hero"	pseudo

and 42, and right-side change events at lines 34, 35, and 38. The left-side change events related to element giant are also included since they are in one composite event that also affects element cast. These change events are collected into one conflict, CB4.

It can be noticed that both conflicts CB3 and CB4 have shared change events. Thus these conflicts have a dependency to each other. It means that if a user chooses to delete giant – choose the left side as the solution – for conflict CB3, the left side change events also have to be selected as the solution for conflict CB4 for consistency. To facilitate computing such dependency, conflicts and change events are designed to have many-to-many relationships as depicted in Figure 4. Thus, if a change event is associated with two or more conflicts, it means that they depend to each others.

It is important to notice that at line 13 in Figure 2 there is a command continue after the addition of a conflict caused by deletion. The command skips the iteration to the next element which avoids unnecessary conflict computation for the current element's features and values. All change events related to the features and values have been collected by the functions getAllRelated*Events(*element*) at lines 5 and 6.

5.5.2 Conflict between Cross-container Moves Lines 15 to 25 in Algorithm 2 are dedicated to identify conflicts related to cross-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 a same container or the element is moved but finally returns to its original container on one of its sides then the conflict is set to PSEUDO. The conflict is then added to conflictList.

Algorithm 3: Algorithm to handle single-valued feature in conflict detection using element tree – handleSingleValuedFeature(*element*, *feature*, *conflictList*) at line 27 in Algorithm 2.

```
   input  : an element element
   input  : a feature feature
   input  : a list to contain conflicts conflictList
 1 begin
       // Handle single-valued feature
       ------------------
 2     if isSingleValued(feature) then
 3         originalValue ← getOriginalValue(feature);
 4         leftValue ← getLeftValue(feature);
 5         rightValue ← getRightValue(feature);
 6         leftEvents ← getAllRelatedLeftEvents(element,
             feature);
 7         rightEvents ← getAllRelatedRightEvents(element,
             feature);
 8         if originalValue <> leftValue or originalValue
             <> rightValue and size(leftEvents) > 0 and
             size(rightEvents) > 0 then
 9             conflict ← createConflict(leftEvents,
                 rightEvents);
10             if leftValue = rightValue or leftValue =
                 originalValue or rightValue =
                 originalValue then
11                 setPseudo(conflict);
12             end
13             addConflict(conflict, conflictList);
14         end
15     end
16 end
```

5.5.3 Single-valued Feature Conflict Conflicts that involve single-valued features are handled by the procedure at line 28 in Algorithm 2, which is elaborated in Algorithm 3. 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 functions getAllRelatedEvents(*element*, *feature*) and getAllRelatedRightEvents(*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 or one of them is equal to originalValue. 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 5. This part of the algorithm also identifies conflict CB5 except that this conflict is set to PSEUDO since both sides change class character's name to a same value, "Hero".

5.5.4 Ordered Multi-valued Feature Conflict Conflicts that involve multi-valued features are handled by the procedure at line 29 in Algorithm 2, which is elaborated in Algorithm 4, where ordered multi-valued features are addressed at lines 3-15. The procedure 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 Figure 6, 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 functions 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 5.

5.5.5 Unordered Multi-valued Feature Conflict Conflict detection for unordered, multi-valued features is handled

Algorithm 4: Algorithm to handle multi-valued feature in conflict detection using element tree – handleMultiValuedFeature(*element*, *feature*, *conflictList*) at line 28 in Algorithm 2.

```
   input  : an element element
   input  : a feature feature
   input  : a list to contain conflicts conflictList
 1 begin
       // Handle multi-valued feature
       -----------------
 2     if isMultiValued(feature) then
 3         if isOrdered(feature) then
 4             values ←
                 getUnequalLeftAndRightValues(feature);
 5             foreach value in values do
 6                 leftEvents ←
                     getAllRelatedLeftEvents(element,
                     feature, value);
 7                 rightEvents ←
                     getAllRelatedRightEvents(element,
                     feature, value);
 8                 if size(leftEvents) > 0 and
                     size(rightEvents) > 0 then
 9                     conflict ← createConflict(leftEvents,
                         rightEvents);
10                     if getLeftIndex(value, feature) =
                         getRightIndex(rightValue, feature)
                         or getLeftIndex(value, feature) =
                         getOriginalIndex(value, feature) or
                         getRightIndex(value, feature) =
                         getOriginalIndex(value, feature)
                         then
11                         setPseudo(conflict);
12                     end
13                     addConflict(conflict, conflictList);
14                 end
15             end
16         else if not isOrdered(feature) then
17             leftValues ←
                 getXORLeftAndOriginalValues(feature);
18             rightValues ←
                 getXORRightAndOriginalValues(feature);
19             values ← leftValues ∪ rightValues;
20             foreach value in values do
21                 leftEvents ←
                     getAllRelatedLeftEvents(element,
                     feature, value);
22                 rightEvents ←
                     getAllRelatedRightEvents(element,
                     feature, value);
23                 if size(leftEvents) > 0 and
                     size(rightEvents) > 0 then
24                     conflict ← createConflict(leftEvents,
                         rightEvents);
25                     if isLeftExisted(value, feature) =
                         isRightExisted(value, feature) or
                         isLeftExisted(value, feature) =
                         isOriginExisted(value, feature) or
                         isRightExisted(value, feature) =
                         isOriginExisted(value, feature)
                         then
26                         setPseudo(conflict);
27                     end
28                     addConflict(conflict, conflictList);
29                 end
30             end
31         end
32     end
33 end
```

at lines 16 to 29 in Algorithm 4. Instead of using function getUnequalLeftAndRightValues, it employs functions getXOR*AndOriginalValues. The functions 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 important in un-ordered features. The procedure to detect a conflict is similar to the procedure for ordered features. The difference is that, to determine if a conflict is PSEUDO or not, it checks the existence of values using functions is*Existed.

6 Evaluation Method

This section presents the method that was employed to evaluate the change-based conflict detection approach proposed in this study and discuss the results. In order to assess the performance benefits of the proposed conflict detection, this study has evaluated it against a mature and widely-used state-based model comparison tool, EMF Compare [2,20], and another implementation of change-based model persistence, EMF Store [17]. EMF Store is not included in the model differencing evaluation in Chapter **??** since it works purely on changes, and it is only designed to identify conflict between changes; not for finding differences between models.

Since there are no manually developed, large models persisted in the proposed change-based format yet, the dataset for this experiments was constructed from a large model reverse-engineered from the Eclipse Epsilon project [21,22]. This model conforms to the Java metamodel [23] and consists of more than 0.54 million elements with a size of 71.1 MBs when persisted in XMI. Compared to sizes of models in model differencing evaluation in Section **??**, the sizes of models in this evaluation is reduced due to slow execution of replaying change events event in EMF Store.

The original model was cloned 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. In the evaluation, 0.44 million artificial changes were applied to each model, generating almost 0.5 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 the proposed change-based format (to be used later in change-based model comparison). After every 20,000 changes, a measurement point is made. The eventual state of the models were persisted in state-based format (to be used later in state-based model comparison) and then conflict detection using EMF Compare, EMF Store, and EMF CBP were performed and their execution time and memory footprint were measured. In one experiment, 22 measurement points to capture their trends. For EMF Store, the changes persisted in EMF CBP format were imported into EMF Store by replaying them in the EMF Store; equivalent changes could be obtained but executed on EMF Store.

This evaluation conducted five experiments to evaluate the model conflict detection of the proposed 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. To reduce the effect of the change on the number of total elements to 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, both operations were still included but their probabilities were made equal so that the number of total elements remain largely unchanged. In the rest of the experiments, homogeneous type change events – isolated from other types – were performed per experiment (e.g. add-only, move-only change events). In the end, 5 results of the experiments were obtained: mixed, add-only, remove-only, move-only, and set-only measurement results. They are useful to asses whether operations of different types have a different impact on model comparison. For the delete-only experiment on EMF Store, due to slow execution of replaying delete event in EMF Store, the size of the models was reduced from 0.54 million to only 39.5 thousand elements each, and the number of changes from 0.44 millions to 33 thousands in 22 measurement points – 1.5 thousand changes for each measurement point.

For conflict detection in EMF CBP, the conflict detection time comprises loading change events, constructing an element tree, and computing conflicts. The memory footprint is the space used to hold the change events, element tree, and conflicts in memory. For EMF Compare, 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. For EMF Store, the conflict detection time comprises loading and mapping change events and computing conflicts. The memory footprint is the space used to hold the change events and mapping and conflicts in memory.

All measurements were performed on the same machine with the following specification: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz (56 processors), 528 GBs main memory, Ubuntu 16.04.6 LTS operating system, and OpenJDK Runtime Environment (build 1.8.0_222-8u222-b10-1ubuntu2 16.04.york0-b10) with JVM InitialHeapSize 2 GBs and MaxHeapSize 32 GBs.

7 Evaluation Results and Discussion

This section reports on the obtained results in terms of comparison time and memory footprint for conflict detection.

7.1 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 Figure **??**. 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 Figure **??**, the total elements remains largely unchanged due to the equal probabilities of addition and deletion as has been set in Section **??**. 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.

Similar to the results in the model differencing evaluation (Figure **??**), the growing number of change events in the conflict detection evaluation is also followed by the logarithmic increase of affected elements (Figure 9a). 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 EMF CBP, EMF Compare, and EMF Store as can be seen in Figure 9b. The differences are due to their distinct conflict detection approaches. EMF Compare detects less conflicts than EMF CBP and EMF Store since its change events are derived, not the real changes. EMF Store detects less conflicts than EMF CBP since its conflicts that depends to each other are grouped into one conflict.

Figure 9c exhibits EMF CBP outperforms EMF Compare and EMF Store in terms of execution time in detecting conflicts, even when the number of change events approaching a million. EMF Store is the worst. It takes more than 35 seconds even though the number of change events has just reached 0.1 millions. Figure 9d also shows EMF CBP outmatches EMF Compare and EMF Store in terms of memory footprint in conflict detection. At the last measurement point, a million change events, EMF CBP only consumes 6 GBs which is much lesser than EMF Compare and EMF Store. EMF Compare occupies around 16 GBs while EMF Store already consumes the same amount of memory footprint only for 0.5 million change events.

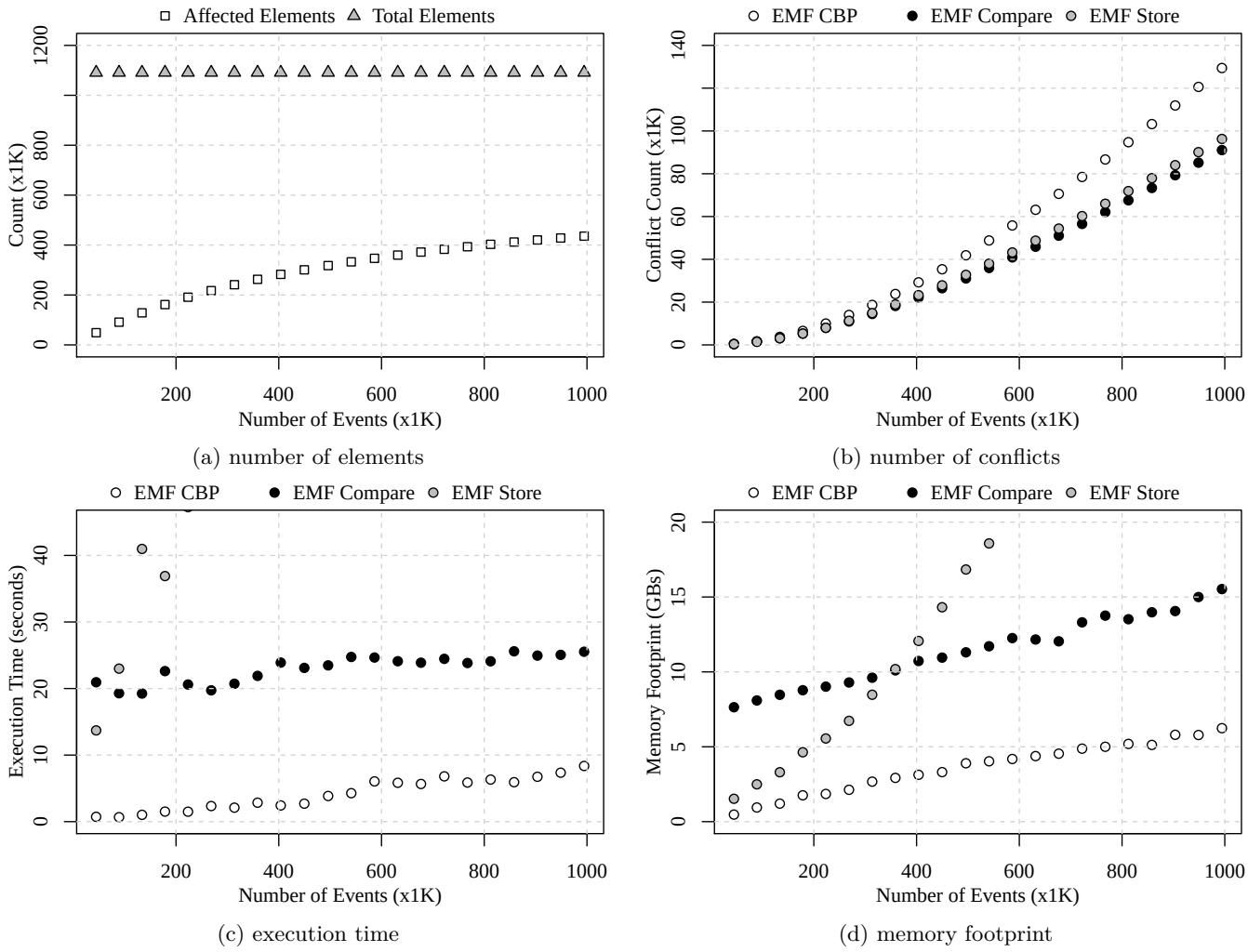
Figures 10, 12, and 14 show the detailed view of EMF CBP, EMF Compare, and EMF Store on the time required to complete conflict detection. As can be seen in Figure 10, the time for EMF CBP to load change events, construct element tree, and detect conflicts grows linearly. In detecting conflicts, the EMF 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 (Figure 12), we can notice that the time taken for matching is less than 5 seconds, and the time used for identifying differences is around 15 seconds in average. 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 differencing tends to be constant since the sizes of the models are set to be as constant as possible (Figure **??**). In contrast, the time for detecting conflicts tends to incline due to the increasing number of conflicting changes as the number of change events increases. In detecting conflicts, EMF Store allocates most of the consumed time for identifying conflicts, and the time increases exponentially. The rest of the time is used for loading changes and mapping them to their affected elements and features (Figure 14).

In terms of memory footprint, EMF CBP allocates most of the memory space for element tree construction and the rest is for the loading change events and identifying conflicts (Figure 11). The reason for this is due to our technical implementation explained in Section **??** (In elementTree, a feature can have many instances even though they refer to the same feature. This causes the memory to increase. One solution is to construct a partial metamodel so that a feature can only have one instance and the instance is used as a key to access the feature's values in each element, similar to the implementation of feature in EMF Framework). In EMF Compare (Figure 13)), 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 (Figure **??**). In contrast, the memory used for detecting conflict increases positively as the number of detected conflicts rises (Figure 9b). For EMF Store, the amount of memory used for loading changes and mapping increases slightly while the amount of memory for identifying conflicts grows exponentially (Figure 15).

7.2 Homogenous Operations

Detection Time. Figure 16 depicts the results of conflict detection time between EMF CBP, EMF Compare, and EMF Store in homogenous operations. The results show that, in all types of homogenous operations, EMF CBP is the fastest on detecting conflicts compared to EMF Compare and EMF Store. The latter has the worst



(a) number of elements

(b) number of conflicts

(c) execution time

(d) memory footprint

Fig. 9: EMF CBP vs. EMF Compare vs. EMF Store comparison as change events increase.

performance in most cases except in delete-only experiment. In this case, EMF Compare is the worst. EMF Compare also requires to calculate dependencies between conflicts. So, when the number of deletion is excessive, EMF Compare performs less efficient than EMF Store (Figure 16b). In the evaluation, this happens when the number of change events exceeds 240 thousands.

Memory Footprint. Figure 17 exhibits the memory footprint resulting from conflict detection in EMF CBP, EMF Compare, and EMF Store with homogeneous operations. The Figure displays that EMF CBP outperforms EMF Compare and EMF Store in terms of memory footprint. EMF CBP only performs worse than EMF Compare in delete-only experiment when the number of change events is more than 80 thousands – model size is 39.5 thousand elements each (Figure 17b. In terms of memory footprint, EMF Store are the worst compared to EMF CBP and EMF Compare. It only performs better than EMF Compare when the number of change events is relatively small – less than 25 thousand change events. In Figure 17c, EMF CBP's memory footprint tends to increase faster than EMF Compare's memory footprint. This is possible since the change events of EMF Compare are actually optimal differences that derived from model differencing which in terms of number is less than real change events recorded in EMF CBP. More random change events means higher possibility to produce more conflicts.

Conflict Count. Figure 18 displays the number of conflicts, both REAL and PSEUDO, detected by EMF CBP, EMF Compare, and EMF Store in the context of homogenous operations. In the add-only experiment as displayed in Figure 18a, all of them detect the same number of conflicts.

Figure 18d shows the results of the change-only experiment. We can notice that the number of conflicts detected by EMF Compare is lower than EMF CBP. This is due to EMF Compare detects no change on an element or feature that has been modified but finally changed back to its original state. While in EMF CBP, this is counted

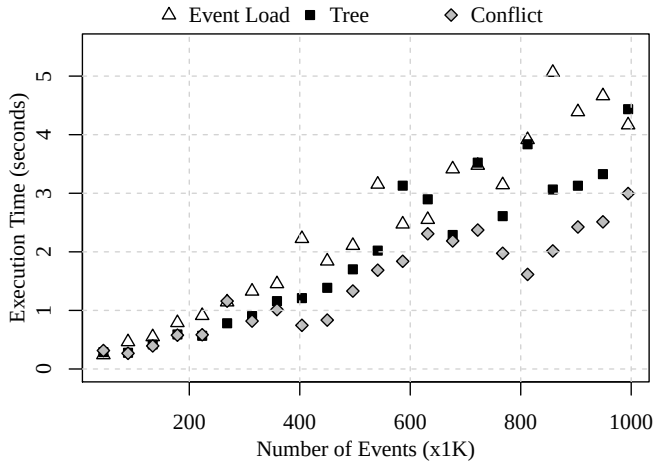


Fig. 10: A breakdown view of EMF CBP on the time required for conflict detection.

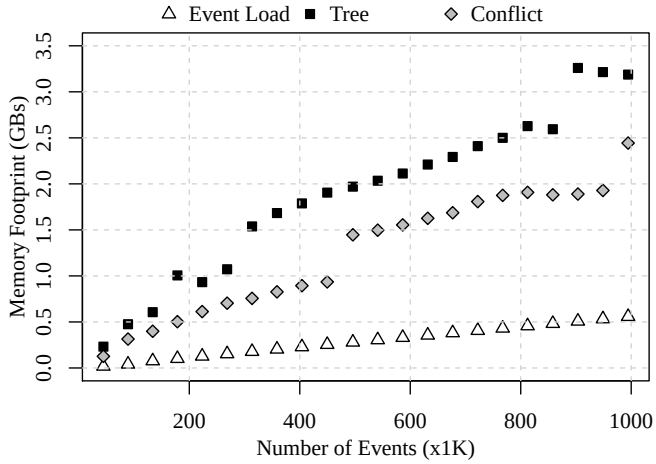


Fig. 11: A breakdown view of EMF CBP on the memory footprint for conflict detection.

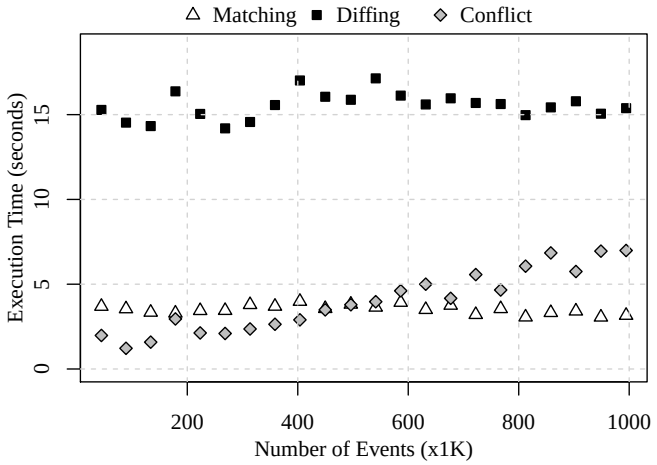


Fig. 12: A breakdown view of EMF Compare on the time required for conflict detection.

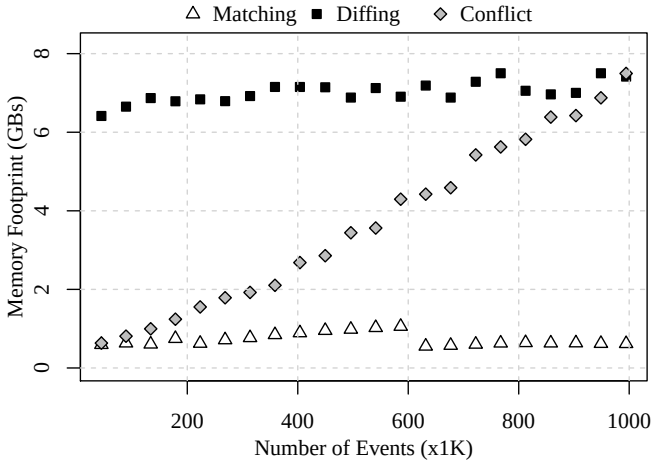


Fig. 13: A breakdown view of EMF Compare on the memory footprint for conflict detection.

as a change with potential to raise a PSEUDO conflict as defined and showed in (12) and Figure 8d. It can also be noticed that the number of conflicts detected by EMF CBP is slightly less than EMF Store detects. It is possible since EMF Store does not consider states in detecting conflicts thus two different change events that are applied to a same element or feature, even though they yield eventual states that are equal to their original state, are considered in conflict.

In the delete-only experiment in Figure 18b, EMF CBP and EMF Compare detect more conflicts than EMF Store since they do not put a conflict that depends on another conflict into one group as performed by EMF Store. For example, the deletion of elements cast and giant, as showed in Tables 1 (EC3 and EC4) and 5 (CB3 and CB4), are separated into two conflicts while EMF Store merges both deletion into one conflict since both are in the same composite event l2 (see conflict ES4 in Table 2). As the number of change events grows, the number of conflicts that share the same change events also increases. Thus, these conflicts are grouped into one conflict causing the number of conflicts decreasing. In addition, EMF CBP detects fewer conflicts than EMF Compare since it skips calculating conflicts for features and values of an element that have been deleted. Change events that affects the features and values have been included when calculating conflicts caused by the deletion of the element as has been explained in the last paragraph of Section 5.5.1. In contrast, EMF Compare treats the conflicts at the features and values of the deleted element as separate conflicts.

Figure 18c shows the results of the move-only experiment. EMF CBP detects more conflicts than EMF Compare since it has change events than EMF Compare. In EMF Compare, its change events are derived and effective which means minimum number of change events are produced. Less number of change events means less probable to produce conflicts and `vice versa`. EMF Store detects less conflicts than EMF CBP and EMF Store due to grouping of conflicts that depend to each other.

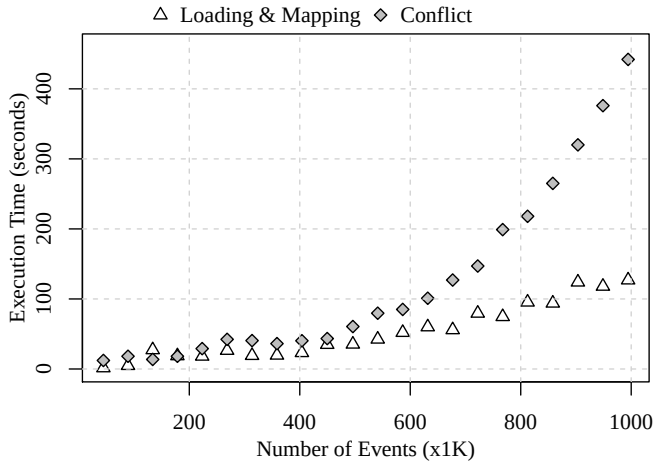


Fig. 14: A breakdown view of EMF Store on the time required for conflict detection.

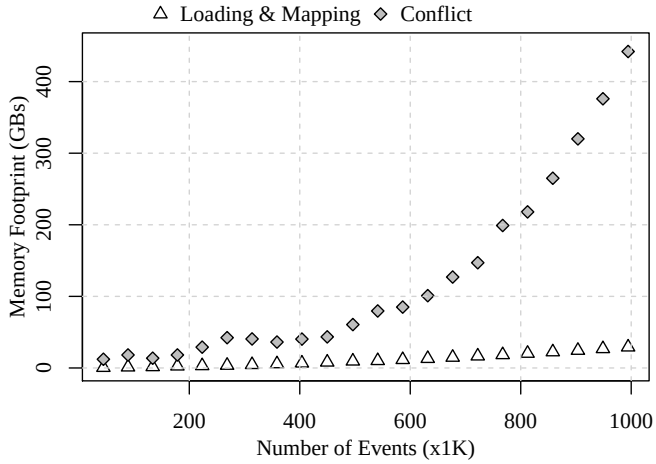


Fig. 15: A breakdown view of EMF Store on the memory footprint for conflict detection.

8 Related Work

The concept of change-based persistence is not new and has been used in persisting changes of software, object-oriented databases, hierarchical documents, and models [4–6, 17].

The nature of change-based persistence give us two advantages. First, it contains finer-granularity information (e.g. types of changes, the order of the changes, elements that were changed, previous values, etc.) of changes which can improve the accuracy of change detection [4–7]. Second, it records changes ordered manner which means that changes made to a model can be identified sequentially without having to explore and compare all elements of compared versions of models [24]. The advantages to detect changes more precisely and much faster can then have positive knock-on effects on supporting (1) developers compare and merge models in collaborative modelling environments [5, 6, 17], and (2) incremental model management [25–27]. Moreover, changed-based persistence contains abundant information which can be exploited for analytics [4].

Nevertheless, change-based persistence also comes with downsides, such as ever-growing model files [4, 24] and increased model loading time [7] which increase storage and computation costs. A model that is frequently modified will increase considerably in file size since every change is added to the file. The increased file size (proportional to the number of persisted changes) will, in turn, increase the loading time of the model since all changes have to be replayed to reconstruct the model's eventual state.

These downsides have to be mitigated to enable the practical adoption of change-based persistence. One approach to reducing the file size of change-based models is by removing changes that do not affect the eventual state of the model. For the increased loading time, it can be mitigated by ignoring – i.e. not replaying – changes that are cancelled out by later changes or employing change-based and state-based persistence side-by-side so that the benefits of state-based persistence on loading time can be obtained.

Other downsides are change-based persistence requires integration with existing tools – since it is still a non-standard approach – for its adoption [17], and still has limited support for standard, text-based version controls for collaborative development [17]. These downsides can be addressed by developing a change-based persistence plugin for a specific development environment (e.g. Eclipse) and persisting changes in text-based format to support text-based version controls (e.g. Git, SVN).

In contrast, state-based persistence has several advantages over change-based persistence. First, since it is the default standard of persisting models, most of the available modelling tools support this kind of persistence thus there is no need for integration with existing tools [17]. Second, it is faster in loading models since there is no need to replay all changes as in change-based persistence. Also, applying lazy loading – elements of models are not loaded upfront – enable faster CRUD (create, read, update, delete) operations [28, 29].

Compared to change-based persistence, state-based persistence has several downsides. First, it is slower than change-based persistence in saving changes [7]. Even thought its backends already implemented lazy loading, it still needs to undergo certain indexing mechanism to persist changes [28–30]. Second, state-based persistence does not have any records of recent elements that have been changed in a model. Thus, every element has to be checked for differences which can be less efficient if the comparison is performed in change-based format [24]. Third, since comparison in state-based format requires deriving differences through a diffing process – not based on actual change records, it can be less accurate than comparison in change-based persistence which is provided

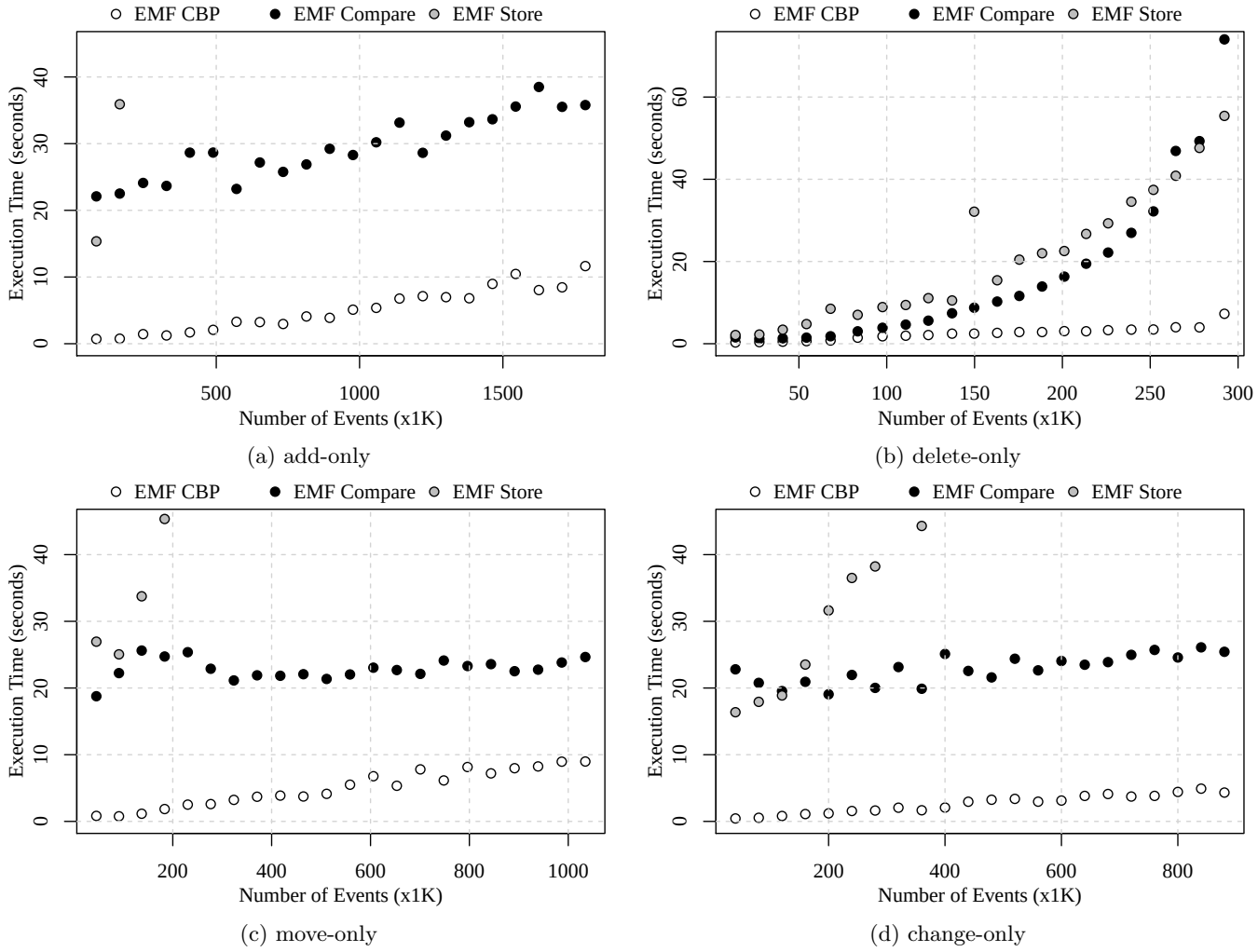


Fig. 16: Conflict detection time for homogeneous operations.

Table 6: The advantages and downsides between change-based and state-based persistence.

Dimensions	Change-based Approach	State-based Approach
Advantages	- Faster for detecting changes [24] - More accurate, carry semantic information [4–7] - Faster and more accurate for comparison and merging [5, 6, 17] - Information carried is useful for analytics [4]	- Faster for loading large models [28–30] - A default standard, no need integration with existing tools [17]
Disadvantages	- Increased record size [4, 24] - Is not efficient for replaying (loading) for long records [7] - Limited supports for standard, text-based version controls (e.g. GitHub) [17] - Not a standard, need integration with existing tools [17]	- Slower for saving changes [7, 28, 29] - Slower for comparison [24] - Less accurate, does not carry semantic information [7, 24]

with more information to detect changes accurately [7,24]. The summary of the advantages and downsides between change-based and state-based persistence are presented in Table 6.

By default, modelling tools that support the 3-layer meta-modelling architectures of Eclipse Modelling Framework (EMF) [31] employ state-based persistence to persist models in Metadata Interchange (XMI) format. XMI is a standard issued by Object Management Group (OMG) for

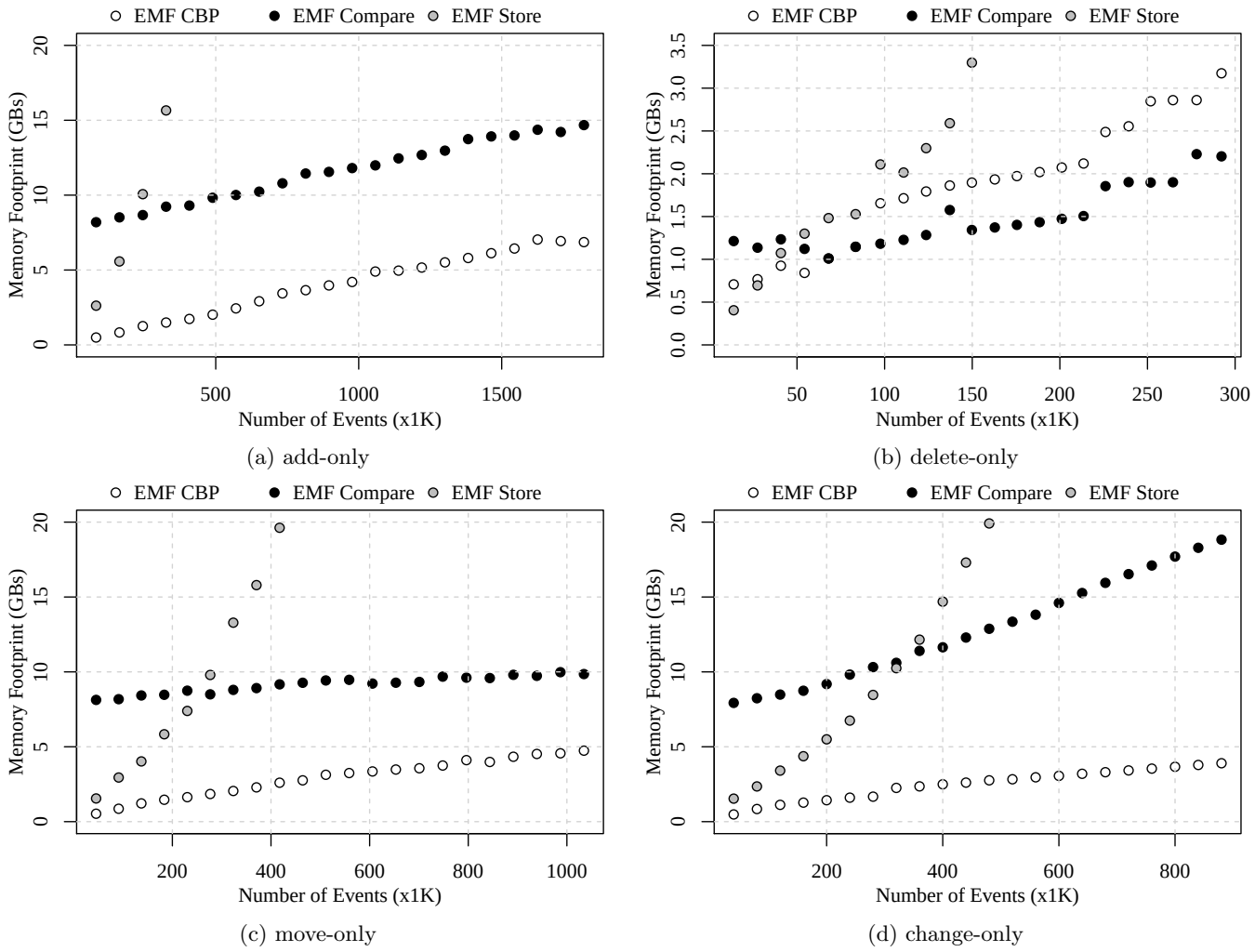


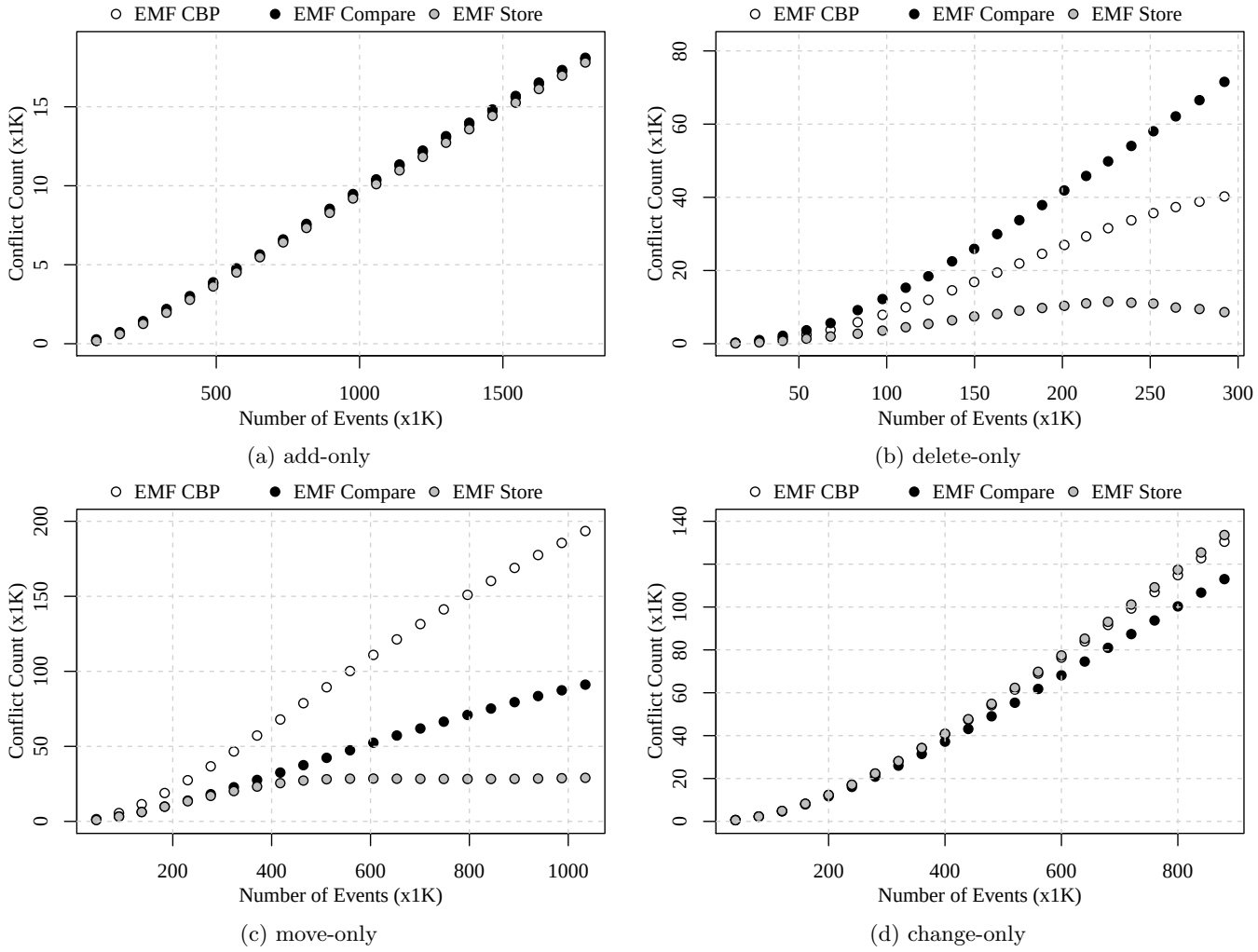
Fig. 17: Conflict detection memory for homogeneous operations.

exchanging metadata information via Extensible Markup Language (XML) [11].

There are several non-XMI approaches to state-based model persistence that use relational or NoSQL databases. For example, EMF Teneo [32] persists EMF models in relational databases, while Morsa [28] persist models in documents with MongoDB backend [33], and NeoEMF [29] persists models in multi NoSQL backends (Graph, Map, Column). However, none of these approaches provides built-in support for versioning and models are eventually stored in binary files/folders which are known to be a poor fit for text-oriented version control systems like Git and SVN. Connected Data Objects (CDO) [30], which provides support for database-backed model persistence, also provides collaboration facilities, but CDO adoption necessitates the use of a separate version control system (e.g. a Git repository for code and a CDO repository for models), which introduces fragmentation and administration challenges [34]. Similar challenges arise in relation to other model-specific version control systems such as EMFStore [17].

The history of diffing can be tracked back to the presence of diff program on Unix or Unix-like platform [35]. The program can perform diffing that is comparing text files "in order to determine how or whether they differ" [36]. Diffing basically is about finding the longest common subsequence between two or more sequences which commonly known as the Longest Common Subsequence (LCS) algorithms [37]. This problem is equivalent to the Shortest Edit Script (SES) problem that is to find the smallest number of editing (add and delete) in order to make a sequence equal to another sequence [38]. LCS or SES algorithms are commonly implemented by Version Control Systems, such as SVN [39] and Git [40], in their diff programs to identify differences between versions of files.

Applying this diffing approach to some non-text artefacts, such as XML [41] and Ecore models [31], is not straightforward since they have different characteristics to text files. For example, XML is a hierarchical document with a tree structure; one node can contains other nodes. The unique feature of XML is that its contain-



(a) add-only (b) delete-only

(c) move-only (d) change-only

Fig. 18: Conflict detection count for homogeneous operations.

ment is unordered whereas in text differencing order is a necessary feature. This has been addressed by Wang et al. [42] by exploiting key XML structure characteristics. Diffing Ecore models is even more complex since the models support multiple characteristics of features, such as attribute/reference, literal/object values, single/multiple values, containment/non-containment, etc [31].

There are several existing tools for model differencing. Beyond EMF Compare [2], which this study used for comparative evaluation due to its maturity and ongoing development activity, tools such as SiDiff [43] and DSMDiff [44] 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 [45]. All of these tools works with state-based persistence to identify differences between models.

This work does not aware of any other work that targets comparison of change-based models persisted in text files. Only EMF Store [17] identified addresses change-based model comparison but it persists models in its own dedicated backend system. Database-backed model persistence and version control solutions such as CDO [30] and EMF Store 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 [46].

9 Conclusions and Future Work

Through persisting models' change history, this research aims at enabling high-performance incremental model processing in collaborative development settings. The pro-

posed approach also has the potential to enable model analytics, more fine-grained tracing, and to improve the precision and performance of model comparison and merging. A prototype implementation of a change-based persistence format has been presented and the main envisioned challenges have been listed.

This work has presented an approach to model differencing by exploiting the nature of change-based persistence which allows us to find differences between versions of a model by only comparing the last set of changes between the two versions.

The change-based model differencing consists of three phases: event loading, element tree construction, and diff computation. In the event loading phase, the implementation loads change events recorded in two change-based model persistence files into memory starting from the line their change events are different. The information contained in the loaded change events are used to construct an element three. An element three essentially is the partial states – only the affected elements and features – of the two versions being compared including the shared original version. It is possible since change events are designed to contain adequate information to construct the element three. A model computation is then executed to identify the differences using a set of predefined rules (i.e., if an element is created in one of the versions it means that the element does not exists in the other version as well as in the original version).

The evaluation results suggest that the the proposed change-based model differencing executes faster than traditional, state-based model differencing. However, the change-based model differencing needs to load change events from a change-based persistence into main memory and thus may requires more memory than for state-based state-based model differencing. In our evaluation, this occurs when the number of change events exceeds 400,000 (around 40 MBs appended change events; not the total size). Arguably, diff and merge operations are usually performed on smaller deltas than this work's evaluation.

Similar to change-based model differencing in the previous research question (RQ2), this work also has proposed an approach to model conflict detection by exploiting the nature of change-based persistence which allows us to detect conflict between versions of a model by only comparing the eventual states of elements and features between the two versions as well as the original version that are affected by the change events.

The phases in the change-based conflict detection is similar to the phases in the change-based model differencing except that the diff computation is replaced with conflict computation. It also consists of a set of rules that compares the eventual states of the elements and features in the element tree as well as the number of change events that affects them in both versions. Asn example, a feature that is modified only in one version cannot have conflicts. A conflict only occurs if the feature is modified on both versions. Also since the the element tree also records every change event to the elements of features that it affect, we can trace change events that cause a conflict.

Based on the findings in conflict detection evaluation, this study found that the proposed change-based model conflict detection approach outperforms the conflict detection approaches in EMF Compare and EMF Store. Nevertheless, models that have been excessively modified or experience significant reduction on model size could impair the performance of the conflict detection as a great number of change records have to be read and loaded into memory.

The proposed change-based model persistence also comes with a number of challenges that this research needs to overcome, such as loading overhead and fast-growing model files. The loading overhead has been addressed in this work by introducing hybrid model persistence – using state and change-based persistence side-by-side – in which models are loaded from its state-based persistence. Nevertheless, the proposed approach still needs to load change events in order to construct an elementTree – Section 5.2 – to perform model differencing and conflict detection, as discussed in Chapters **??** and **??**. The loading can be further optimised to consume less memory and speed up parsing, such as using binary or optimised-text format.

The fast-growing model files challenge has not been addressed in this work. Persisting models in a change-based format means that model files will keep growing in size during their evolution significantly faster than their state-based counterparts. This work proposes two solutions to address the issue: (1) sound change-compression operations (e.g. remove older/unused information) that can be used to reduce the size of a model in a controlled way, (2) a compact textual format that will minimise the amount of space required to record a change (a textual line-separated format is desirable to maintain compatibility with file-based version control systems).

The information contained in change-based model persistence is also useful for Model Analytics. With appropriate tool support, modellers will be able to "replay" (part of) the change history of a model (e.g. to understand design decisions made by other developers, for training purposes). In state-based approaches, this can be partly achieved if models are stored in a version-control repository (e.g. Git). However, the granularity would only be at the commit level. By analysing models serialised in the proposed representation, modelling language and tool vendors will be able to develop deeper insights into how modellers actually use these languages/tools in practice and utilise this information to guide the evolution of the language/tool. By attaching additional information to

each session (e.g. the id of the developer, references to external documents/URLs), sequences of changes can be traced back to the developer that made them, or to requirements/bug reports that triggered them.

In the future evaluation, once the plugin of change-based model persistence has been established, this work intends to evaluate it against state-based model persistence on complex models in real-world scenarios.

References

1. D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, ser. CVSM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/CVSM.2009.5071714
2. EMFCompare, "Emf compare developer guide," https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html, accessed: 2018-11-01.
3. B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, Sep. 2003.
4. R. Robbes and M. Lanza, "A change-based approach to software evolution," *Electr. Notes Theor. Comput. Sci.*, vol. 166, pp. 93–109, 2007. [Online]. Available: https://doi.org/10.1016/j.entcs.2006.06.015
5. E. Lippe and N. van Oosterom, "Operation-based merging," in *SDE 5: 5th ACM SIGSOFT Symposium on Software Development Environments, Washington, DC, USA, December 9-11, 1992*, 1992, pp. 78–87. [Online]. Available: http://doi.acm.org/10.1145/142868.143753
6. C. Ignat and M. C. Norrie, "Operation-based merging of hierarchical documents," in *The 17th Conference on Advanced Information Systems Engineering (CAiSE '05), Porto, Portugal, 13-17 June, 2005, CAiSE Forum, Short Paper Proceedings*, 2005. [Online]. Available: http://ceur-ws.org/Vol-161/FORUM_17.pdf
7. T. Mens, "A state-of-the-art survey on software merging," *IEEE Trans. Software Eng.*, vol. 28, no. 5, pp. 449–462, 2002. [Online]. Available: https://doi.org/10.1109/TSE.2002.1000449
8. 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
9. A. Yohannis, H. H. Rodriguez, F. Polack, and D. Kolovos, "Towards efficient comparison of change-based models," B. Combemale and A. Shaukat, Eds., vol. 18, no. 2, Jul. 2019, pp. 7:1–21, the 15th European Conference on Modelling Foundations and Applications. [Online]. Available: http://www.jot.fm/contents/issue_2019_02/article7.html
10. EMFStore, "What is EMFStore and why should I use it?" https://www.eclipse.org/emfstore/, accessed: 2019-08-15.
11. OMG, "About the XML Metadata Interchange Specification Version 2.5.1," http://www.omg.org/spec/XMI, accessed: 2018-02-21.
12. EpsilonLabs, "emf-cbp," https://github.com/epsilonlabs/emf-cbp, accessed: 2019-06-06.
13. F. Polack, A. Yohannis, H. Hoyos Rodriguez, D. Kolovos *et al.*, "Towards visualising change-based models," in *Models and Evolution Workshop@ Models 2019*. Keele University, 2019.
14. 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
15. ——, "Towards hybrid model persistence," in *Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 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
16. M. Koegel, M. Herrmannsdoerfer, O. von Wesendonk, and J. Helming, "Operation-based conflict detection," in *Proceedings of the 1st International Workshop on Model Comparison in Practice*, ser. IWMCP '10. New York, NY, USA: ACM, 2010, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1826147.1826154
17. 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
18. eclipse, "emf-store," https://git.eclipse.org/c/emf-store, accessed: 2019-08-21.
19. 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
20. Eclipse, "EMF Compare," https://www.eclipse.org/emf/compare/, accessed: 2018-01-15.
21. ——, "Epsilon Git," http://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/commit/?id=ebd0991c279a1f0df1acb529367d2ace5254fe87, accessed: 2018-02-19.
22. ——, "Epsilon," https://www.eclipse.org/epsilon/, accessed: 2018-02-12.
23. ——, "Java Metamodel," https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.java.doc%2Fmediawiki%2Fjava_metamodel%2Fuser.html, accessed: 2019-01-08.
24. M. Koegel, M. Herrmannsdoerfer, Y. Li, J. Helming, and J. David, "Comparing state- and operation-based change tracking on models," in *Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2010, Vitória, Brazil, 25-*

29 October 2010, 2010, pp. 163–172. [Online]. Available: https://doi.org/10.1109/EDOC.2010.15

25. F. Jouault and M. Tisi, "Towards incremental execution of atl transformations," in *Theory and Practice of Model Transformations*, L. Tratt and M. Gogolla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 123–137.
26. B. Ogunyomi, L. M. Rose, and D. S. Kolovos, "Property access traces for source incremental model-to-text transformation," in *Modelling Foundations and Applications - 11th European Conference, ECMFA 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-24, 2015. Proceedings*, 2015, pp. 187–202. [Online]. Available: https://doi.org/10.1007/978-3-319-21151-0_13
27. I. Ráth, Á. Hegedüs, and D. Varró, "Derived features for EMF by integrating advanced model queries," in *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, 2012, pp. 102–117. [Online]. Available: https://doi.org/10.1007/978-3-642-31491-9_10
28. J. Espinazo-Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, 2011, pp. 77–92. [Online]. Available: https://doi.org/10.1007/978-3-642-24485-8_7
29. G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot, "Neoemf: A multi-database model persistence framework for very large models," *Science of Computer Programming*, vol. 149, pp. 9 – 14, 2017, special Issue on MODELS'16. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642317301600
30. Eclipse, "Eclipse CDO The Model Repository," https://www.eclipse.org/cdo/documentation/, accessed: 2019-04-02.
31. 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
32. Eclipse, "Teneo," http://wiki.eclipse.org/Teneo, accessed: 2017-10-15.
33. MongoDB, "The database for modern applications," https://www.mongodb.com/, accessed: 2019-07-23.
34. K. Barmpis and D. S. Kolovos, "Evaluation of contemporary graph databases for efficient persistence of large-scale models," *Journal of Object Technology*, vol. 13, no. 3, pp. 3: 1–26, 2014. [Online]. Available: https://doi.org/10.5381/jot.2014.13.3.a3
35. J. Hunt and M. MacIlroy, *An algorithm for differential file comparison*, ser. Computing science technical report. Bell Laboratories, 1976. [Online]. Available: https://books.google.co.uk/books?id=zJ2LMwAACAAJ
36. O. Dictionary, "diff," https://www.lexico.com/en/definition/diff, accessed: 2019-07-23.
37. L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, Sep. 2000, pp. 39–48.
38. 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
39. svn, "svn diff (di," http://svnbook.red-bean.com/en/1.8/svn.ref.svn.c.diff.html, accessed: 2019-07-23.
40. git, "Git - git-diff Documentation," https://git-scm.com/docs/git-diff, accessed: 2019-07-23.
41. W. W. W. Consortium, "Extensible Markup Language (XML)," https://www.w3.org/XML/, accessed: 2019-07-24.
42. Y. Wang, D. J. DeWitt, and J. . Cai, "X-diff: an effective change detection algorithm for xml documents," in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, March 2003, pp. 519–530.
43. 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
44. 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
45. 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.
46. 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.