Change-Based Model Differencing and Conflict Detection

Alfa Ryano Yohannis

ary506@york.ac.uk

Supervisors:

Dimitris Kolovos

Fiona Polack

Horacio Hoyos Rodriguez

Department of Computer Science

University of York

United Kingdom

December 30, 2019

A Thesis Submitted in Partial Fulfilment of the Requirements

for the Degree of Doctor of Philosophy in Computer Science

"Thank you, Mum!"

Abstract

In large-scale computer systems and software development, model-driven engineering is an approach that focuses on the development and management of models. The models are usually expressed in diagrams, textual notations, or code. Most of these models persist in state-based formats. While state-based persistence has certain advantages, it is problematic when it comes to detecting changes in large-scale models. As an alternative, this work proposes a change-based approach that involves persisting the full sequence of changes made to models. Persisting a model in a change-based format has the potential to deliver benefits over state-based persistence, such as the ability to perform model differencing and conflict detection much faster and more precisely. This can then yield positive follow-on effects to help developers compare and merge models in collaborative modelling environments. Nevertheless, change-based persistence also comes with downsides, including increased model loading time.

This work investigates two approaches to reduce loading time. The first is to identify and ignore superseded changes, and the second uses hybrid model persistence. While the first approach is still greatly outperformed by state-based persistence, the hybrid model persistence experiences only a slight slowdown in most cases compared to loading models from state-based persistence.

This work proposes an approach for faster model differencing and conflict detection. It works by exploiting the nature of change-based persistence, which allows us to find differences and conflicts between two versions of a model by comparing only the last set of changes applied to them, without having to compare every element and feature in both versions as is traditionally done in state-based model comparison. This work's evaluation shows that the proposed change-based model differencing and conflict detection outperform the existing traditional state-based approach. Nevertheless, models that have been excessively modified or experience significant reductions in model size could impair the performance of the proposed model differencing and conflict detection as numerous change records must be read and loaded into memory.

Contents

Abstract 1

Contents 4

List of Figures 6

List of Tables 8

Listings 10

1 Introduction 13

1.1 Background 13

1.2 Research Aim 15

1.3 Research Objectives 15

1.4 Research Outputs 15

1.5 Research Scope 16

1.6 Thesis Structure 16

1.6.1 Chapter 2: Literature Review 16

1.6.2 Chapter 3: Analysis and Hypothesis 17

1.6.3 Chapter 4: Designing Change-based Persistence for Models . 17

1.6.4 Chapter 5: Optimised Loading of Change-based Models . . . 17

1.6.5 Chapter 6: Hybrid Model Persistence 18

1.6.6 Chapter 7: Efficient Model Differencing of Change-based Models 18

1.6.7 Chapter 8: Efficient Conflict Detection of Change-based Models 19

1.6.8 Chapter 9: Conclusions and Future Work 19

1.7 Publications 19

2 Literature Review **21**
2.1 Models in This Research . . . 21
2.2 Model Persistence . . . 22
2.2.1 Text Files . . . 22
2.2.2 Relational Databases . . . 23
2.2.3 NoSQL Databases . . . 23
2.2.4 Change-based Representation (EMF Store) . . . 24
2.2.5 Change-based vs. State-based Persistence . . . 28
2.3 Model Differencing and Conflict Detection . . . 30
2.3.1 The Challenges of Model Comparison . . . 32
2.3.2 Identifying Changes in Models . . . 33
2.4 Conclusions . . . 34

3 Analysis and Hypothesis **35**
3.1 Summary of Findings . . . 35
3.2 Hypothesis and Research Questions . . . 37
3.3 Research Method . . . 38
3.4 Conclusions . . . 40

4 Designing Change-based Persistence for Models **41**
4.1 Introduction . . . 41
4.2 Running Example: Part I . . . 42
4.3 Proposed Approach . . . 44
4.4 Prototype Implementation . . . 45
4.5 Challenges . . . 50
4.6 Conclusions . . . 51

5 Optimised Loading of Change-based Model Persistence **53**
5.1 Introduction . . . 53
5.2 Running Example . . . 54
5.3 Toward Efficient Loading of Change-Based Models . . . 56
5.3.1 Model History . . . 56
5.3.2 Set and Unset Events . . . 58
5.3.3 Add, Remove, and Move Events . . . 59

5.3.4 Create and Delete Events . . . 61
5.4 Evaluation . . . 61
5.4.1 Data Description . . . 63
5.4.2 Model Loading Time . . . 64
5.4.3 Model Saving Time . . . 65
5.4.4 Memory Footprint . . . 67
5.4.5 Discussion . . . 68
5.5 Conclusions . . . 69

6 Hybrid Model Persistence 71
6.1 Introduction . . . 71
6.2 Comparing Change- and State-based Model Persistence . . . 72
6.3 Hybrid Model Persistence . . . 73
6.4 Implementation . . . 74
6.5 Evaluation . . . 77
6.5.1 Storage Space Usage . . . 77
6.5.2 Time and Memory Footprint of Loading and Saving Models . 78
6.6 Discussion . . . 80
6.7 Conclusions . . . 80

7 Efficient Model Differencing of Change-based Models 82
7.1 Introduction . . . 82
7.2 Running Example: Part II . . . 83
7.3 State-based Model Differencing . . . 86
7.4 Change-based Model Differencing . . . 89
7.4.1 Event Loading . . . 90
7.4.2 Element Tree . . . 90
7.4.3 Diff Computation . . . 99
7.5 Evaluation . . . 104
7.5.1 Method . . . 105
7.5.2 Results and Discussion . . . 106
7.6 Conclusions . . . 111

8 Efficient Conflict Detection of Change-based Models 113

8.1 Introduction . . . 113
8.2 State-based Conflict Detection (EMF Compare) . . . 114
8.3 Change-based Conflict Detection (EMF Store) . . . 118
8.3.1 Summary . . . 121
8.4 EMF CBP Conflict Detection . . . 122
8.4.1 Change Event Mapping . . . 122
8.4.2 Theoretical Foundation . . . 123
8.4.3 Conflict Computation . . . 126
8.5 Accuracy of Conflict Detection . . . 133
8.5.1 EMF CBP vs. EMF Compare . . . 133
8.5.2 EMF CBP vs. EMF Store . . . 134
8.6 Evaluation Method . . . 136
8.7 Evaluation Results and Discussion . . . 138
8.7.1 Mixed Operations . . . 138
8.7.2 Homogeneous Operations . . . 142
8.7.3 Conclusions . . . 147

9 Conclusions and Future Work 148
9.1 Research Questions Addressed . . . 148
9.2 Limitations and Validity . . . 152
9.3 Future Work . . . 153
9.4 The Big Picture . . . 154

Bibliography 157

Appendix A An Example of Change-based Model Persistence 166

List of Figures

2.1 An example to show how EMF Store works. 25

4.1 An excerpt of the UML-like meta-model of the running example in Figure 4.2. 42
4.2 Three incomplete class diagrams of a Role Playing Game. 42
4.3 Event classes to represent changes of models. 47
4.4 Factory, resource, and ChangeEventAdapter classes. 49

5.1 Running example of a meta-model and a conformant model. 54
5.2 CBP workflow, with optimised loading elements indicated by starred blocks. 56
5.3 The class model defining Model History. 57
5.4 The object diagram of the CBP model history in Listing 5.4. 57
5.5 Results for loading a model in original CBP (CBP) and optimised CBP (OCBP) and for loading a state-based (XMI) representation. . 64
5.6 A comparison of the time required to persist an event between original CBP (CBP), optimised CBP (OCBP), and XMI. 66
5.7 A comparison of the memory footprint after loading a model by original CBP (CBP), optimised CBP (OCBP), and XMI. 66
5.8 A comparison of the memory footprint after persisting an event by CBP, optimised CBP, and XMI. 68

6.1 The mechanism of hybrid model persistence. 73
6.2 Class diagram showing the core components of the hybrid model persistence implementation. 75

7.1 A comparison of the left and right models in Listings 4.2 and 4.3. . . 89
7.2 A class diagram showing the core components of the change-based approach to speed up model differencing and conflict detection. . . . 91
7.3 An element tree constructed from information in CBPs in Listing 7.2 (left change events only). 92
7.4 An element tree constructed from information in CBPs in Listings 7.2 and 7.3 (all left and right change events). 95
7.5 Steps in Element Tree construction. 97
7.6 total elements, affected elements, and diffs 106
7.7 Change-based vs. state-based model differencing as differences increase. 107
7.8 Breakdown view of comparison time and memory footprint in Figure 7.7. 108
7.9 Comparison time for homogeneous operations. 109
7.10 Memory footprint for homogeneous operations. 110

8.1 Conflicting and non-conflicting change events (dashed arrow = left change event, solid arrow = right change events, circle = state). . . . 125
8.2 Changes in EMF CBP, EMF Compare, and EMF Store as change events increase. 140
8.3 Detailed view of EMF CBP on the time required for conflict detection. 141
8.4 Detailed view of EMF CBP on the memory footprint for conflict detection. 141
8.5 Detailed view of EMF Compare on the time required for conflict detection. 141
8.6 Detailed view of EMF Compare on the memory footprint for conflict detection. 141
8.7 Detailed view of EMF Store on the time required for conflict detection. 141
8.8 Detailed view of EMF Store on the memory footprint for conflict detection. 141
8.9 Conflict detection time for homogeneous operations. 143
8.10 Conflict detection memory for homogeneous operations. 144
8.11 Conflict detection count for homogeneous operations. 145

List of Tables

2.1 Advantages and downsides of different model persistence solutions. . 27

2.2 The advantages and downsides of change-based and state-based persistence. 29

5.1 Description of change-based models generated for evaluation. 63

5.2 The t-test results of loading time by original CBP (CBP), optimised CBP (OCBP), and XMI. 64

5.3 The t-test results of saving time by original CBP (CBP), optimised CBP (OCBP), and XMI. 65

5.4 The t-test results of the memory footprint after loading a model by original CBP (CBP), optimised CBP (OCBP), and XMI. 67

5.5 The t-test results of the memory footprint from saving an event by original CBP (CBP), optimised CBP (OCBP), and XMI. 68

6.1 Comparison of model persistence approaches. 72

6.2 Space usage for the Epsilon and BPMN2 projects and the Wikipedia article on the United States (m = million events, MB = Megabytes, KB = Kilobytes). 78

6.3 A comparison of the time and memory footprint for loading and saving models of the hybrid and state-based-only persistence. Time is in seconds, and the memory footprint is in MB. 79

6.4 Hybrid model persistence compared to other persistence approaches. 80

8.1 Conflicting change events identified by EMF Compare based on the case in Figure 4.2. 117

8.2 Conflicting change events identified by EMF Store in Listings 7.3 and 7.2. 120
8.3 The advantages and drawbacks of EMF Compare and EMF Store in detecting conflicts. 121
8.4 Mapping the elements, features, and values in Figure 7.4 to the events that affect them. 123
8.5 Conflicting change events in Listings 7.2 and 7.3 identified by the proposed change-based conflict detection. The bold identifiers are the keys where conflicts were detected. 127

Listings

4.1 Simplified XMI file of the original version in Figure 4.2a. 43
4.2 Simplified XMI file of the left version in Figure 4.2b. 43
4.3 Simplified XMI file of the right version of Figure 4.2c. 44
4.4 The complete change events of Bob's model in Figure 4.2b. 46
4.5 Simplified Java code to handle notification events. 48
4.6 An example how to use CBPResource in Java code. 50
5.1 State-based representation in simplified XMI of the tree model in Figure 5.1b. 55
5.2 State-based representation in simplified XMI of the tree model in Figure 5.1c. 55
5.3 Change-based representation of the tree model in Figure 5.1b. 55
5.4 Change-based representation of the tree model in Figure 5.1c. 55
5.5 A CBP representation of attribute name assignments ended with SET. 58
5.6 A CBP representation of attribute name assignments ended with UNSET. 58
5.7 A CBP of add and remove operations. 59
5.8 A CBP representation of add, move, and remove operations. 59
5.9 A naïve optimised CBP representation of original CBP representation in Listing 5.8 60
5.10 Change-based representation of the model in Figure 5.1 after removal of node n3. 61
7.1 Change-based representation of the original version in Figure 4.2a. . 83
7.2 The appended events made by Bob to produce Figure 4.2b. 84
7.3 The appended events made by Alice to produce Figure 4.2c. 85
7.4 Diffs presented as change events. 88

8.1 The derived, minimal change events to produce the left version (Bob's version) in Figure 4.2b from the original version (Jane's version). . . 115
8.2 The derived, minimal change events to produce the right version (Alice's version) in Figure 4.2c from the original version (Jane's version). 115
A.1 Change-based representation of the model in Figure 4.2b. 166

Acknowledgements

I would like to thank my supervisor, Professor Dimitris Kolovos, for providing this opportunity and for his endless support and advice throughout this PhD. I would also like to thank my second supervisor, Professor Fiona Polack, for her help and comments, especially during the early stages of this PhD. Also, I am very grateful to my third supervisor, Dr Horacio Hoyos Rodriguez, for all his insightful comments during his time at the University of York.

I would further like to thank all the members of the Enterprise Systems research group for their helpful comments on my thesis. Thank you to Antonio García-Domínguez and Ran Wei who did the preliminary work on EMF CBP. Thank you to Nicholas Matragkas, my internal examiner, who always asks me challenging questions to prepare me for the viva.

The achievements of this thesis would not have been possible without the support of my parents, who provided me with the opportunity to pursue my doctorate. It would not be possible also without the help of my brothers during my time working on the project. Thank you to all my mates, Kang Allyn, Jandy, Micky, Dibyo, Teny, Nungky, and others. It was precious spending time with you here in York sharing our PhD stories.

This work was partly supported through a scholarship managed by *Lembaga Pengelola Dana Pendidikan Indonesia* (Indonesia Endowment Fund for Education). Without that scholarship, this thesis would not be possible.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Chapter 1

Introduction

This chapter briefly presents the background of the research presented in this thesis and the aim of this research. Several research objectives to accomplish the aim of the research are then defined, followed by a discussion of research outputs and scoping. Finally, this chapter presents the structure of this thesis and lists the papers that have been published from this research.

1.1 Background

In large-scale computer systems and software development, model-driven engineering is an approach that focuses on the development and management of models—usually expressed in diagrammatic or textual notations. Most of these models persist in state-based formats. In a state-based format, 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 expensive for state-based models [1]. The research presented in this thesis is motivated by the need to find a more efficient approach to model differencing and conflict detection.

As an alternative to state-based persistence, this work proposes that a model can be persisted in a change-based format, which persists the full sequence of *changes*

made to the model. The concept of change-based persistence has been used in persisting changes to software, object-oriented databases, and hierarchical documents [2–4]. Change-based approaches facilitate detection of differences between versions, and they make better semantic identification of the differences. They do this by providing information with finer granularity (e.g. types of changes, the order of the changes, elements that were changed, and previous values). Better and more-granular identification of differences can provide better support for resolution of conflicts, e.g. where versions of a model have been modified in different ways [5]. The ordered nature of change-based persistence means that changes made to a model can be identified sequentially without having to explore and compare all elements of the model against its previous version. The ability to detect changes faster and with precision can then have positive follow-on effects to support (1) model differencing, conflict detection, and merging in collaborative modelling environments, and (2) incremental model management (e.g., incremental query [6] and model-to-text transformation [7]). Based on these arguments, this work explores the advantages and shortcomings of change-based persistence as an alternative approach to state-based persistence for models conforming to three-layer meta-modelling architectures such as the Eclipse Modelling Framework (EMF) [8] and Meta-Object Facility (MOF) [9].

Nevertheless, change-based persistence also comes with downsides, such as ever-increasing model files [2, 10] and increased model loading time [5], which increase costs for storage and computation. Every time a model is modified, the file that records its list of changes increases in size. The increased file size (proportional to the number of persisted changes), in turn, increases the loading time of the model since all changes must be replayed to reconstruct the model's eventual state. These downsides need to be mitigated to enable the practical adoption of change-based persistence. Another downside is that change-based persistence requires integration with existing tools for its adoption [11], since it is still a non-standard approach. This downside can be addressed by developing a change-based persistence plugin for a specific development environment (e.g. Eclipse).

1.2 Research Aim

The aim of this work is to assess the advantages and shortcomings of a novel change-based approach to model persistence against existing textual and database-backed state-based and database-backed changed-based model persistence formats. This work is concerned with models that conform to meta-models expressed in object-oriented meta-modelling languages such as Ecore and MOF. The advantages and shortcomings considered in this work are in terms of computational cost and memory usage for 1) model loading, 2) model saving, 3) model differencing, and 4) conflict detection.

1.3 Research Objectives

This research has defined the following research objectives to accomplish the aim of the research.

1. Identify and study existing model persistence approaches in the context of the EMF meta-modelling architecture.

2. Identify and study change-based artefact persistence approaches beyond EMF.

3. Design a generic change-based model persistence format for models that conform to arbitrary EMF (Ecore) meta-models and to implement algorithms for saving and loading models in that format.

4. Implement algorithms for differencing and conflict detection between two versions of change-based models.

5. Assess the performance and memory use of loading, saving, differencing, and conflict detection of change-based models against established model persistence approaches within EMF.

1.4 Research Outputs

By the end of this research, the following outputs have been produced:

1. Prototypes of change-based persistence, change-based model differencing, and

change-based conflict detection.

2. Designs and evaluation results of novel approaches for loading time reduction and saving, model differencing, and model conflict detection of change-based models.

3. Publications [12–15] and a thesis documenting the solutions proposed in this research.

1.5 Research Scope

The scope of this research is as follows:

1. This work is restricted to models that conform to three-level meta-modelling architectures. The Eclipse Modelling Framework is used as a representative of such architectures for the implementation of all solutions and prototypes.

2. This work only covers change-based model persistence, differencing, and conflict detection. Change-based model merging is beyond the scope of the research presented in this thesis.

3. Although it is mentioned several times in this report, the use of change-based persistence to support incremental model management is not part of this work.

1.6 Thesis Structure

This section provides an overview of the remaining chapters of the thesis.

1.6.1 Chapter 2: Literature Review

This chapter summarises work related to change-based persistence and comparison, critically assesses the advantages and disadvantages of current approaches, and seeks opportunities to contribute new knowledge to the field. The chapter comprises a brief discussion on methods to identify changes in models, the benefits of change-based solutions to model management, the advantages and drawbacks of state- and change-based persistence in the context of software engineering, the state of the art in model persistence, the state of the art in model comparison, state-based model

differencing, state-based conflict detection, change-based conflict detection, and the research method applied in this work.

1.6.2 Chapter 3: Analysis and Hypothesis

This chapter summarises on the findings of the literature review and presents the motivation for a new change-based persistence format and a novel approach to improve the performance of model differencing and conflict detection by exploiting change-based persistence. Based on the findings in Chapter 2, this chapter presents the hypothesis and research questions addressed in this study. It also presents an overview of the research method used to answer the research questions.

1.6.3 Chapter 4: Designing Change-based Persistence for Models

This chapter presents the concept of the change-based model persistence proposed in this research and its prototype implementation. Its contents have been published in a workshop [12].

1.6.4 Chapter 5: Optimised Loading of Change-based Models

Change-based persistence comes with the downside of ever-growing file sizes [2, 10], which causes increased loading time [5]. Reducing the loading time is essential to facilitate the practical adoption of change-based persistence. One way to reduce loading time is by ignoring—not replaying—changes that are cancelled out by subsequent changes.

To evaluate the efficiency of the proposed approach, an optimised loading algorithm that ignores superseded change events is compared to a naïve loading of a change-based representation and loading the same model from a state-based representation. They are compared on the time required to load the models and their memory footprints. Evaluation is also performed on the time required for persisting changes between change-based and state-based persistence. The contents of this chapter are based largely on a published conference paper [13].

Compared to the naïve change-based representation, the optimised version shows considerable savings in terms of loading time and a negligible impact on saving time, but at the cost of a higher memory footprint. However, in terms of loading time and

memory footprint, XMI outperforms both approaches, but it is much less efficient in saving changes.

1.6.5 Chapter 6: Hybrid Model Persistence

While optimised loading is faster than naïve loading, the benefits are moderate, and optimised loading is still slower than loading from a state-based representation [14]. This finding has motivated the design and development of a hybrid approach to persistence that augments a change-based representation with a fully derived, state-based representation.

The hybrid model persistence approach is evaluated by comparing it to state-based persistence (e.g. XMI, NeoEMF [16]) in terms of time, the memory footprint, and the storage space required to load models and persist changes. An evaluation is also performed of the time required to detect changes between hybrid and state-based persistence. The contents of this chapter are based largely on a conference paper [14].

Results of the evaluation indicate that the hybrid approach to model persistence provides benefits on model loading time, since its performance is comparable to loading a model from a change-based persistence only, with trade-offs on increased memory footprint and storage space usage.

1.6.6 Chapter 7: Efficient Model Differencing of Change-based Models

This chapter describes change-based model differencing and its implementation with an evaluation. Change-based persistence is expected to speed-up model differencing because the information required to identify changes is already contained in the models' persistence.

The proposed model differencing is evaluated by comparing it to state-based model differencing in terms of the time and memory footprint required to find all differences between two versions of a model. The contents of this chapter are based largely on a workshop paper [15].

Based on our experiments, this study argues that the change-based comparison approach works best for large models that have been modified a moderate number of

times. Our experiments demonstrate savings in the order of 90% for (relatively) small changes made to large models. However, models that have been excessively modified and experience a significant reduction of model size could impair the performance of change-based model differencing as a high number of change records must be read and loaded into memory.

1.6.7 Chapter 8: Efficient Conflict Detection of Change-based Models

This chapter presents change-based model conflict detection. After identifying the differences between two versions of a change-based model, this work also aims to detect conflicts between two versions of a model. Model conflict detection is a crucial step that precedes model merging.

Similar to change-based model comparison, the proposed conflict detection also is evaluated by comparing it to the conflict detection of existing change- and state-based persistence in terms of the affected time and memory footprint.

The findings from the conflict detection evaluation indicate that the proposed approach can substantially reduce conflict detection time (up to more than 90% in some experiments) compared to existing state-based and change-based conflict detection approaches. Nevertheless, models that have been excessively modified or that experience a significant reduction in model size could impair the performance of the conflict detection, as a great number of change records must be read and loaded into memory.

1.6.8 Chapter 9: Conclusions and Future Work

This chapter summarises the work that has been carried out and uses the results of the evaluations to answer the research questions and hypothesis proposed in Section 3.2. It also states limitations and threats to the validity of the findings of this thesis and suggests future work to address them.

1.7 Publications

The research in various parts of the thesis has been published in the following papers:

1. A. Yohannis, D. S. Kolovos, and F. Polack, 'Turning models inside out,' in Proceedings of MODELS 2017 Satellite Events co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017., 2017, pp. 430–434. [Online]. Available: http://ceur-ws.org/Vol-2019/flexmde_8.pdf (Chapter 4).

2. A. Yohannis, H. H. Rodriguez, F. Polack, and D. S. Kolovos, 'Towards efficient loading of change-based models,' in Modelling Foundations and Applications—14th European Conference, ECMFA 2018, held as Part of STAF 2018, Toulouse, France, June 26–28, 2018, Proceedings, 2018, pp. 235–250. [Online]. Available: https://doi.org/10.1007/978-3-319-92997-2_15 (Chapter 5).

3. A. Yohannis, H. H. Rodriguez, F. Polack, and D. S. Kolovos, '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 (Chapter 6).

4. 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 (Chapter 7).

Chapter 2

Literature Review

This chapter presents the literature review of this study. First, it highlights key characteristics—unique features, strong points, and downsides—of some existing implementations. It then summarises the advantages and drawbacks of the two main types of persistence: state-based and change-based persistence, and it introduces desirable characteristics for a new change-based persistence implementation. This chapter then reviews the related work on identifying differences and detecting conflicts between versions of models. It then presents the challenges that model differencing and conflict detection are currently dealing with, as well as the downsides of existing approaches to solving the problems, which gives motivation to this research to come up with a new solution. Last, the conclusions of the literature review are presented.

2.1 Models in This Research

A model is an abstract representation of an entity [17]. It can be used for different purposes: as a sketch to communicate a system, as a blueprint to define the specification of a system, or as a modifiable artefact to generate a working system [18]. In model-based software engineering, the latter is the scenario in which models are mainly used. In that scenario, a model is created using a modelling language, and the model should conform to its meta-model—an abstraction that describes the model. Later, the model can be transformed to generate a software artefact through model transformation/code generation [19]. The software artefact, its model, and

the model's meta-model create a three-layered abstraction which is known as the three-layer meta-modelling architecture.

Eclipse Modeling Framework (EMF) [20] is a technical implementation of such an architecture. It is a framework and code-generating facility that allows developers to define meta-models, create models, and generate implementations of the models [20]. In this research and literature review, we focus on models and modelling tools that support the three-layer meta-modelling architecture of EMF.

2.2 Model Persistence

In constructing models, modelling tools should be able to support model persistence so that models under construction can be saved at any time and reloaded for further modification. Most tools persist models in a state-based format. That is, they capture a snapshot of a model at a time and then persist its entire state into storage. The model state can be persisted in different forms, such as text files, relational databases, or NoSQL databases.

2.2.1 Text Files

The simplest and most common way to save a model is to persist it into a text file. By default, modelling tools that support the three-layer meta-modelling architectures of Eclipse Modeling Framework (EMF) [20] persist a model in a text file with a format of Metadata Interchange (XMI)—a standard issued by Object Management Group (OMG) for exchanging metadata information using Extensible Markup Language (XML) [21].

Since it is the default for persisting EMF models, it is supported by most modelling tools. To modify a model persisted in an XMI file, such as performing create, read, update, delete (CRUD) operations, a tool has to de-serialise and load the model from the file into memory. This can be a problem when we want to make a few changes but the size of the model is very large—it takes considerable time and memory to load the model. Also, when it is saved, the model must be persisted in its entirety. This is not efficient when we made only a few changes. Since it is a text-based file, the model can be duplicated and shared with minimum effort, e.g. through manual copy or

version control systems (e.g. Git [22] and SVN [23]). However, for model differencing (see Section 2.3), text-based differencing [24] cannot be applied accurately to XMI files since they are essentially tree documents which require different differencing approaches [25].

2.2.2 Relational Databases

Models can also be persisted into relational databases. EMF Teneo [26] is a solution that integrates EMF with existing persistency solutions, such as Hibernate [27] and EclipseLink [28]. Thus, it can persist EMF models into relational database backends. In this way, EMF Teneo can utilise the power of storage, caching, and querying of the database backends. It also supports the automatic mapping of models to relational model schema with flexible mapping customisation. Using relational databases as its backends enables EMF Teneo to support the lazy loading of models. So, when performing CRUD operations, it only loads and saves relevant elements and features—not the entire model—into and from memory. This is efficient in terms of memory usage.

Similar to EMF Teneo, Connected Data Objects (CDO) [29] also supports persisting models into various database elements model persistence (e.g. relational and NoSQL databases). It is a development-time model and meta-model repository as well as a distribution and runtime persistence framework for EMF-based application systems. It supports model versioning and can perform model differencing and conflict detection—it uses EMF Compare [30] to perform the comparison [31]. One downside of CDO is that it requires the use of a separate version control system (e.g. a Git repository for code and a CDO repository for models). This can introduce fragmentation and create challenges to file administration [32].

2.2.3 NoSQL Databases

In the era where data are abundant and models are getting larger, the ability to handle large models is necessary. Tools, such as Morsa [33] and NeoEMF [16], have been developed to persist models into non-relational (NoSQL) databases. Morsa saves models in documents with MongoDB as its backend [34], while NeoEMF persists models in multiple NoSQL backends: Neo4j [35] for Graphs, MapDB [36]

for Maps, and Apache HBase [37] for Column datastores. The advantages of using NoSQL databases are that users are given options to choose which datastores—with some degree for configuration—that best fit the characteristics of their models and meta-models. This helps to maximise the features the backends provide, such as lazy loading and caching. Neither Morsa nor NeoEMF 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.

2.2.4 Change-based Representation (EMF Store)

All the solutions previously mentioned persist models in state-based formats. EMF Store [11] takes a different approach; it persists models in a change-based representation. EMF Store appears to be the only current implementation of change-based persistence for EMF models.

EMF Store is a model repository, and it supports collaborative editing and versioning of models [38]. Instead of using standard text-oriented version controls (e.g., Git, SVN) for model versioning, EMF Store has its own dedicated, change-based, model-oriented versioning mechanism. Models are shared through a server and distributed to client applications. Clients can modify the models in parallel, offline or online, and synchronise with the server. Conflicts caused by concurrent modification are detected automatically, and they can be resolved interactively by users. The historical changes to models are kept on the server, and different versions of a model as well as changes that produced them can be retrieved from the server.

In EMF Store, to version models, a project must first be created. A project can contain one or more models. Every project has a version history, and each version represents a commit of a client. A commit sends a package of changes to the server. The package itself contains a collection of operations that transforms the project to a newer version or can be expressed as the differences (the deltas) between the two versions. An operation can be add, delete, set, unset, or move that modifies an element or feature, or it can be a composite operation—one that consists of many operations, e.g. re-factoring, which moves a method to a superclass.

To obtain a specific version of an existing project, a client performs a checkout. This

version is called the base version on the client side. The client can then modify this version. Every operation applied to the version is recorded by EMF Store. When the client commits, these operations are put into one package and sent to the server. If the base version is still the head version of the project on the server, the commit is accepted, and a new version is created. If the base version is not the head version, it means that another client has committed its changes to the server. Thus, the current client has to synchronise it by updating its local project. This is the state where conflicts can happen between the incoming version and local changes, that is, when they modify the same element or feature of a model. EMF Store performs conflict detection to identify conflicts automatically. The mechanism of EMF Store to identify conflicts is discussed in detail in Section 8.3.

As an illustration to show how EMF Store works, let's say that Jane's created a project on the server (Figure 2.1, step 1) setting an initial version, v0, of the model. She also shared it so that her team members could also work on the same project. Jane then created an initial model (step 2) and committed it to the server (step 3). As she committed her work on the server, operations o1 and o2 recorded while she was creating the initial model were also sent to the server producing version v1.

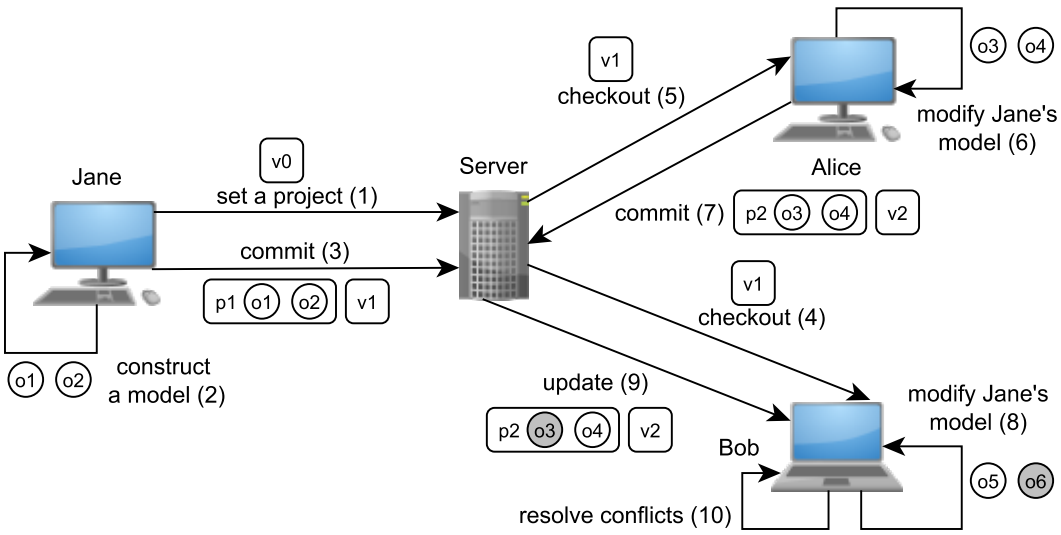


Figure 2.1: An example to show how EMF Store works.

Bob and Alice then checked out Jane's work from then server creating copies of version v1 on their local machines (steps 4 and 5). Alice edited Jane's model by performing operations o3 and 04 (step 6). She then committed her to the server,

producing version v2 (steps 7). Her commit was straightforward since her base version v1 was the same as on the server — no conflict detection is needed. After the commit, the server hold three versions of the model, v0, v1, and v2, including with their packages of operations, p1, containing o1 and o2, and p2, containing o3 and o4.

Bob also modified Jane's model in parallel. He performed operations o5 and o6 (step 8). However, when he tried to commit his work, he was required to update his work (step 9) since his base version was different from the one that was on the server due to the previous commit performed by Alice. His base version was v1, while v2 was the latest version on the server. When updating, a conflict detection was performed to detect conflicts between the server's operations and his local operations. If there was a conflict, he was required to solve the conflict first before he could commit his work to the server (step 10).

The primary motivation from EMF Store to a change-based approach is that calculating the differences between two versions in state-based persistence can be expensive and less accurate [39] (State-based model differencing identifies differences using LCS algorithms [24, 30], not the real changes). Since it follows a change-based approach, EMF Store does not store the state of every version. It saves operations of each version in an ordered manner only so they can be executed and reversed to obtain the states between versions. Nevertheless, it also stores the intermediate cached states for selected versions, including the head version, to speed up the retrieval of specific versions.

The advantages of EMF Store are that it was designed to allow semantic versioning of models. It can make model differencing and conflict detection more accurate and efficient rather than they are on state-based model persistence [39]. By default, the packages of operations are persisted in XMI files, but EMF Store can also be configured to use other backends like MongoDB [40]. The downsides of EMF Store are that it has its own mechanism for controlling versions. This limits its adopters to use common text-oriented version controls [41], such as Git and SVN. Its performance can also degrade as more models/users are added to a repository [42].

The advantages and downsides of the different model persistence solutions presented in this section can be found in Table 2.1. These advantages and downsides reveal

Table 2.1: Advantages and downsides of different model persistence solutions.

Product	Advantages	Downsides
XMI	+ default standard, widely supported + easy to duplicate and share by manual copy or text-oriented version controls	– requires loading the entire model to modify – a model is saved in its entirety – supports text-oriented version controls, but applying text-based differencing might produce inaccurate results
Teneo	+ supports lazy loading, only load and save affected elements and features when performing CRUD operations + can be supported by database backends: rollback, caching, etc.	– does not support model versioning, comparison, and merging – multiple concurrent accesses can cause a bottleneck – poor fit for text-oriented version controls since models are persisted in database
CDO	+ supports lazy loading, only load and save affected elements and features when performing CRUD operations + supports model versioning, comparison, and merging + can be supported by database backends: rollback, caching, etc.	– fragmentation and administration challenges because of separation of version controls between models and code – poor fit for text-oriented version controls since models are persisted in database
Morsa & NeoEMF	+ supports lazy loading, only load and save affected elements and features when performing CRUD operations + can be supported by NoSQL backends: handling big data, graph data, etc.	– do not support model versioning, comparison, and merging – poor fit for text-oriented version controls since models are persisted in database
EMF Store	+ supports semantic versioning of models, which allows model merging and conflict detection to be more effective + faster in detecting conflicts when numbers of changes are relatively small	– requires loading the entire model to modify – a model is saved in its entirety even for small changes – persists models in the forms of files/folders and using its own mechanism for model versioning; thus, it is a poor fit for text-oriented version controls

points to consider on how to design a change-based model persistence format that is compatible with version control systems such as Git and SVN. These considerations are presented in Section 3.1.

2.2.5 Change-based vs. State-based Persistence

This section compares the advantages and drawbacks of change-based and state-based persistence in general, not limited to EMF models. Change-based persistence works by persisting the complete change history of an artefact instead of persisting a snapshot—the entire state—of an artefact at a time. The concept of change-based persistence is not new; it has been used to persist changes of software, object-oriented databases, hierarchical documents, and models [2–4, 11].

Change-based persistence offers two main advantages. First, it records information (e.g. types of changes, the order of the changes, elements that were changed, and previous values) with finer granularity. This can improve the accuracy of change detection [2–5]. Second, it records changes in an ordered manner, which means that changes made to an artefact can be identified sequentially without having to explore and compare all elements of compared versions of an artefact [10]. The advantages to detect changes more precisely and much faster can then have related benefits: (1) developers can compare and merge artefacts in collaborative environments [3, 4, 11] and (2) incremental management [6, 7, 43]. Moreover, changed-based persistence contains a wealth of information which can be exploited for analytics [2].

Nevertheless, change-based persistence also comes with downsides, such as ever-growing artefact files [2, 10] and increased artefact loading time [5], which increases storage and computation costs. An artefact 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 artefact since all changes must be replayed to reconstruct the artefact's eventual state.

Other downsides are that change-based persistence requires integration with existing tools—since it is still a non-standard approach—for its adoption [11], and it still has limited support for standard, text-based version controls for collaborative development

[11]. 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 summary, state-based persistence has several strong points. First, since it is the default standard persistence approach for most artefacts, it requires minimum effort to integrate with existing tools [11]. Second, it is faster in loading artefacts persisted in state-based format since there is no need to replay all changes as with change-based persistence. Also, some artefacts support lazy loading. For example, an artefact is not loaded in its entirety upfront. Only parts affected by an operation are loaded into memory. This enables faster CRUD (create, read, update, delete) operations [16, 33].

Table 2.2: The advantages and downsides of change-based and state-based persistence.

Dimension	**Change-based Approach**	**State-based Approach**
Advantages	+ More accurate, carries semantic information [2–5] + Faster and more accurate for detecting changes, comparison, and merging [3, 4, 11] + Information carried is useful for analytics [2]	+ Faster for loading large artefacts [16, 29, 33] + A default standard, no need to integrate with existing tools [11]
Disadvantages	– Increased record size [2, 10] – Not efficient for replaying (loading) long records [5] – Limited support from standard, text-based version controls (e.g. GitHub) [11] – Not a standard, needs integration with existing tools [11]	– Slower for saving changes (XMIs) [5, 16, 33] – Slower for comparison [10] – Less accurate, does not carry semantic information [5, 10]

Compared to change-based persistence, state-based persistence also has downsides. First, it is slower than change-based persistence in saving changes [5]. For an artefact persisted in state-based format and does not support lazy loading, the artefact must be persisted in its entirety even though only a single change has been made. Second, state-based persistence does not keep records of changes to an artefact. Thus, every part of the artefact must be checked for differences. This can be less efficient if the comparison is performed in a change-based format [10]. Third, comparison in a state-based format requires identifying differences through a diffing process—not

based on actual change records. So, it can be less accurate than a comparison in change-based persistence which is provided with more information to detect changes accurately [5, 10]. The advantages and downsides of change-based and state-based persistence are summarised in Table 2.2.

2.3 Model Differencing and Conflict Detection

The history of model differencing and conflict detection can be traced back to the presence of the diff program on Unix or Unix-like platforms [44]. Diffing is a function that compares text files 'to determine how or whether they differ' [45]. It is commonly known as the Longest Common Subsequence (LCS) algorithm [46], and it is equivalent to the Shortest Edit Script (SES) problem: finding the smallest number of edits (adds and deletes) to make a sequence equal to another sequence [24]. LCS or SES algorithms are commonly implemented by Version Control Systems, such as SVN [47] and Git [48], in their diff programs to identify differences between versions of files.

Using diffing on graph-based artefacts, such as XML [49] and Ecore models [20], is not straightforward since they are different from text files. For example, XML is a hierarchical document with a tree structure; one node can contain other nodes. The unique feature of XML is that its containment is unordered, whereas in text files differencing order is a necessary feature. This has been addressed by Wang et al. [25] by exploiting key XML structure characteristics. Identifying differences between Ecore models is even more complex since those models support multiple characteristics of features, such as attribute/reference, literal/object values, single/multiple values, and containment/non-containment [20].

There are several existing tools for model differencing. EMF Compare [30] is a popular tool for comparing and merging EMF models, with generic support for different meta-models. It is an extensible framework, so it can be adapted to the specific needs of certain meta-models. EMF Compare works by matching elements of the models being compared and then executing differencing to identify the differences between them. Matching and differencing are discussed in detail in Chapter 7. EMF DiffMerge (EDM) [50] is similar to EMF Compare except that its abstraction is at a lower level, and it is designed to prevent data loss and enforce model consistency [51].

As a consequence, EMF Compare could use the EDM engine when it needs to enforce a particular consistency policy. Also, it supports scoping, which means that the comparison does not must be at the model level. It could also be applied to sets of model elements—subsets of models—that can be defined arbitrarily by using specific filters [52]. In this study, EMF Compare is used as a baseline for evaluation because of its maturity and ongoing development activity. Other tools, such as SiDiff [53] and DSMDiff [54], 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 [55]. All of these tools work with state-based persistence to identify differences between models.

Our literature review has not identified any other work that targets comparison of change-based models persisted in text files. Only EMF Store [11] addresses change-based model conflict detection, but it persists models in its own dedicated backend system. Moreover, since it is designed to identify conflicts between changes, it does not give direct, summarised information about which parts of two versions of a model are different—not for model differencing. It only gives lists of changes to users. The summarised information is useful in the scenario where a model is excessively changed in both versions since users do not have to interpret the long lists to identify differences between the versions. Moreover, it works only on changes; it does not consider eventual states of models in detecting conflicts [56]. Thus, if an element has been changed concurrently, but the changes produce eventual states that are equal to their original state, EMF Store still treats these changes as if they were in conflict. Database or dedicated-backend model persistence and version control solutions such as CDO [29] and EMF Store provide model conflict detection capabilities between different versions of the same model, but they present integration challenges when users wish to use text-oriented version control systems (e.g. Git, SVN) which are typically file-based. Moreover, their performance can degrade as more models/users are added to a repository [42].

2.3.1 The Challenges of Model Comparison

Identifying differences between versions of models can become crucial for large evolving models, particularly in the later phases of the development cycle when many small changes are made to fine-tune the models [57]. This challenge has been addressed by incremental model management where changes to models are recorded and used as the basis for effective incremental model processing operations. Egyed [58] has shown that the property-access recording approach is applicable to query such changes. More recent work has shown that variants of this approach can be used to achieve incrementality in a wide range of model processing operations, including model-to-model transformation [43], model-to-text transformation [7], model validation, and pattern matching [6]—as long as the changes can be precisely identified.

Nonetheless, this approach works best at identifying differences between serial versions of a model; it is not as straightforward in identifying differences between parallel—branched—versions. In addition, the solutions in incremental model management are coupled with their execution engines. This means they work best in single-developer environments. (This is discussed further in Section 2.3.2). In a collaborative setting, as the size and complexity of a model grows, it is common to manage the model in multiple parallel versions. Thus, the ability to identify differences between parallel versions and to detect conflicts between the differences is very important.

Model differencing and conflict detection must be executed before two versions of a model are merged. However, performing model differencing and conflict detection in the typical state-based approach is computationally expensive and memory-greedy. (This is discussed further in Section 2.3.2). In traditional, state-based model comparison, every element of the versions being compared must be loaded into memory, matched, and then differenced [30]. This is inefficient for large models that undergo only a few changes. A novel approach is required that can compare only elements that have been modified – —not all elements—to speed up model comparison.

2.3.2 Identifying Changes in Models

There are two approaches in the literature for identifying changes in models: using notification facilities and model differencing. These are reviewed inf the sections that follow.

Notifications

In this approach, a model change tracking engine must hook into the notification facilities of the modelling tool used to edit the model, so that the engine can receive notifications as soon as a change happens (e.g. class Giant has been deleted, class Character has been renamed to 'Hero'). This is an approach taken by the IncQuery incremental pattern matching framework [6] and the ReactiveATL incremental model-to-model transformation engine [7]. The main advantage of this approach is that precise and fine-grained change notifications are provided for free by the modelling tool. (They do not need to be computed by the execution engine—which as discussed below can be expensive and inefficient). On the downside, this approach is a poor fit for collaborative development settings where modelling and automated model processing activities are performed by different members of the team.

Model Differencing

This approach eliminates the coupling between modelling tools and model change tracking engines. Instead of depending on live notifications, in this approach the developer needs to have access to a copy of the last or other version of the model, so it can be compared against the current version of the model (e.g. using a model-differencing framework such as EMF Compare [30] or EMF DiffMerge [50]) and the differences (the delta) can be computed on demand. The main advantage of this approach is that it works well in a collaborative development environment where typically developers have distinct roles and responsibilities. On the downside, model comparison and differencing are computationally expensive and memory-greedy as both versions of the model must be loaded into memory before they can be compared.

In summary, tracking changes in models using notification facilities currently delivers significant performance benefits only in a single-developer environment as the approach is coupled to modelling tools. As a result, in collaborative development en-

vironments, developers must either forgo the notification approach altogether or work with model differencing, which is computationally expensive and memory-greedy.

2.4 Conclusions

This chapter presented a review of literature in the areas of model persistence and differencing. It summarised the advantages and drawbacks of state-based and change-based model persistence, and related work on identifying differences and detecting conflicts between versions of models.

Chapter 3

Analysis and Hypothesis

This chapter summarises the findings of the literature review and presents the motivation to develop a new change-based persistence format and a novel approach to improve model differencing and conflict detection by exploiting change-based persistence. Based on the findings in Chapter 2, this chapter presents the hypothesis and research questions addressed in this study. It also presents an overview of the research method used to answer the research questions.

3.1 Summary of Findings

Performing model differencing and conflict detection in state-based persistence can be expensive in terms of computation time [10]. This is because state-based model differencing requires every element of the two versions being compared to be inspected, matched, and diffed to identify their differences [30]. Even persisting state-based models using database backends—such as in Teneo [26], CDO [29], Morsa [33], and NeoEMF [16]—can reduce only the overhead cost of loading models, since all elements still need to be checked. Imagine if we have made only small changes on a model, but all of its elements must be examined to identify differences. This approach is not efficient and can become a bottleneck, especially in collaborative environments where models are often managed in different concurrent versions. Differencing, conflict detection, and merging are common in that context.

As an alternative to state-based persistence, change-based persistence has the po-

tential to deliver high-performance model differencing and conflict detection since the change history of a model is already contained in the model's change-based representation [3, 4, 11]. Therefore, identifying changes through model differencing is not required as in state-based persistence. Moreover, model differencing and conflict detection in change-based persistence can also be more accurate than performing them in state-based persistence since the persistent representation also contains detailed information, such as the order of changes, types of changes, and elements affected by changes [2–5].

So far, we have identified EMF Store as the only implementation of change-based model persistence that conforms to the Eclipse Modeling Framework (EMF). However, this research did not use and extend EMF Store for several reasons. First, EMF Store is a full-fledged client-server model repository and versioning system. This means that it requires a certain degree of administration activities (e.g. server configuration, user authentication and authorisation), and it creates a dependency on EMF Store. We favour avoiding such administration activities and dependency and prefer a solution that can version on shared models through different text-oriented version controls (e.g. SVN, Git). Second, it does not scale up well. There is performance degradation as more models/users are added to a repository and models grow in size as discussed in [42] and as evidenced by our own evaluation in Sections 7.5 and 8.7. Third, EMF Store detects conflicts between changes that produce different states when merging. However, it cannot be used directly for model differencing. It is not designed to identify differences between two versions of a model. Fourth, it works only on changes and does not consider eventual states of models in detecting conflicts [56]. As a consequence, if an element has been changed concurrently, but the changes produce eventual states that are equal to their original state, EMF Store still treats these changes as though they are in conflict. Last, EMF Store is in maintenance mode. That is, there is no active feature development going on, and its end-of-life might be declared in 2022 [38].

Based on these considerations, we aimed for a new change-based persistence for EMF-based models. Such an implementation should be able to capture and persist all the changes of models into text-based files, and it should be able to exploit the persisted changes to produce high-performance model differencing and conflict

detection.

3.2 Hypothesis and Research Questions

The research in this thesis aims to improve model differencing and conflict detection. Based on the literature review, change-based model persistence has the potential to deliver such performance. To assess whether change-based persistence can improve model differencing and conflict detection, the following hypothesis has been established **'a textual change-based model persistence approach can outperform existing model persistence formats in terms of model saving, model differencing, and conflict detection time, with an overhead in terms of model loading time and memory use'**.

In this thesis, the word 'model' refers to typed object graphs that conform to three-layer object-oriented meta-modelling architectures such as Eclipse Modeling Framework (EMF) [8].

Model differencing is used to identify the differences between versions of a model, such as determining what has been changed from an original version of a model or comparing versions of a model created by different teams working independently. The main goal of conflict detection is to ascertain whether independent updates can be merged, or whether there are conflicts (elements or features that differ in ways that are incompatible) that must first be resolved.

'Execution time' as used in the hypothesis is the time required to perform model saving, model differencing, or model conflict detection. We are particularly interested in the benefits and the challenges of using change-based persistence for large models; these are models having more than a million elements as per [16, 33]. Model loading time is the time required to load a model from its persistent representation into memory. Memory use is the size of the memory occupied during model saving, loading, differencing, and conflict detection.

To assess the validity of the hypothesis, this work aims to answer the following research questions:

1. **How can models be persisted in a change-based format, and how does change-based persistence perform, compared to state-based persistence, in terms of loading and saving models? (RQ1)**

   The concept of change-based persistence must be translated into an implementation in a modelling framework context so it can be applied to model persistence, so that its impact on model loading and saving, and later model differencing and model conflict detection can be assessed.

2. **In a change-based format, how can the differences between models be identified, and how does change-based model differencing perform, in terms of speed and memory footprint, compared to state-based model differencing? (RQ2)**

   One of the main motivations for exploring the use of change-based persistence is to speed up model differencing. Because of the nature of change-based persistence, the mechanism to perform change-based model differencing will differ substantially from current state-based model differencing approaches. It is expected that model differencing in change-based persistence will perform faster than model differencing in state-based persistence.

3. **Following change-based model differencing, how can conflicts be detected between versions of a model, and how does change-based conflict detection perform, in terms of speed and memory, compared to state-based model conflict detection? (RQ3)**

   The follow-on effects of change-based persistence on model conflict detection will also be investigated. It is expected that conflict detection of change-based models will be significantly faster than conflict detection of state-based models.

3.3 Research Method

In performing this research, this research follows the experimental process proposed by Wohlin et al. [59]. The experimental process comprises five activities: scoping, planning, operation, analysis and interpretation, and presentation and packaging.

Scoping. In the scoping activity, the hypothesis, goals, and objectives of an experiment must be defined [59]. Basili et al. [60] provide the following questions (scoping points) in their framework to help determine the scope of an experiment in software engineering: (1) what is studied? (object of study), (2) what is the intention? (purpose), (3) which effect is studied? (quality focus), (4) whose view? (perspective), and (5) where is the study conducted? (context).

Planning. In the planning activity, these components must be defined: context selection, hypothesis formulation, selection of variables, selection of subjects, experiment design selection, instrumentation, and validity evaluation [59]. The context can be offline vs. online, student vs. professional, toy vs. real problems, specific vs. general. Hypotheses have to be stated, and the data gathered throughout the experiment should be used – using appropriate statistical tests – to reject or accept the hypotheses. The independent and dependent variables to be measured must be determined. The subjects must represent the case being studied so the results of the experiment can be generalised. The experiment must be designed carefully to get the desired results, and suitable standard design types should be selected. Experiment objects, guidelines, and measurement instruments also should be defined to ensure the experiment is executable. Last, validity threats should be identified and evaluated.

Operation. The operation activity comprises three steps: preparation, execution, and validation [59]. In the preparation, all the materials needed for the experiment are selected and prepared. The experiment can be executed in several ways, such as once or on multiple occasions, for one year or several years. Execution requires that the experiment is on the right track, not interrupted, and running correctly. Validation means that the data produced must be reasonable and collected orderly.

Analysis and interpretation. Descriptive statistics and visualisation can be used to understand the data. Unnecessary data and variables can be removed to facilitate analysis and interpretation. Hypothesis testing is used to reject or accept the experiment's hypothesis. The analysis and interpretation should explain how the data gathered contribute to the rejection or acceptance of the hypothesis. The results might be statistically insignificant, but the lessons might still be worth learning [59].

Presentation and packaging. In this activity, the experiment's results should be documented and published in research papers so they are available to other researchers. The experiment also should be packaged to support other parties who wish to replicate it [59].

3.4 Conclusions

Chapter 2 presented the advantages, downsides, and challenges of current approaches to model persistence, differencing, and conflict detection in the scientific literature. In this chapter, we have pointed out design considerations that any proposed solution should deliver to achieve high-performance model differencing and conflict detection. From there, we established the hypothesis and research questions of this study. Finally, we presented an overview of the research method used in this research.

Chapter 4

Designing Change-based Persistence for Models

This chapter presents a novel approach to change-based model persistence, including it format, requirements, design, and implementation. Potential benefits and novel capabilities as well the challenges of using a change-based format for model persistence also are highlighted in this chapter using a running example.

4.1 Introduction

The concept of change-based persistence presented in the literature review must be translated for a modelling framework if it is to be applied for model persistence. To gain all the benefits of change-based persistence, an implementation that can save and load a model in change-based persistence must be developed first. The implementation should be able to capture all relevant changes of a model and persist them into a file. It must also be able to de-serialise changes from the file and (re)execute them in order to (re)construct the model. This research has developed a prototype of such a tool, designed to work with EMF models and meta-models.

Before exploring how change-based persistence is implemented, this chapter introduces a running example to explain the solutions proposed in this study and how model differencing and conflict detection are performed in existing tools, such as in EMF Compare [30] and EMF Store [38]. This example is used throughout this thesis to

explain the proposed solutions as well.

The rest of this chapter is structured as follows. Section 4.2 introduces the running example. Sections 4.3 presents an overview of the proposed approach. Section 4.4 discusses the prototype implementation on top of the Eclipse Modeling Framework. The challenges of change-based model persistence are presented in Section 4.5. Section 4.6 concludes this chapter.

4.2 Running Example: Part I

Figure 4.2 shows three versions of an incomplete model conforming to a simplified UML-like meta-model in Figure 4.1. The meta-model is minimalist to facilitate explaining the running example.

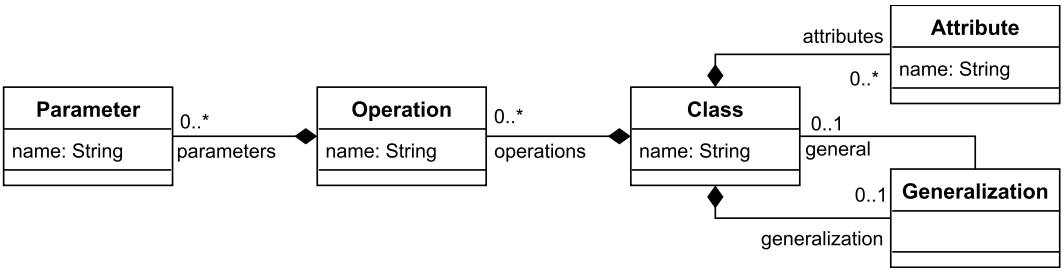
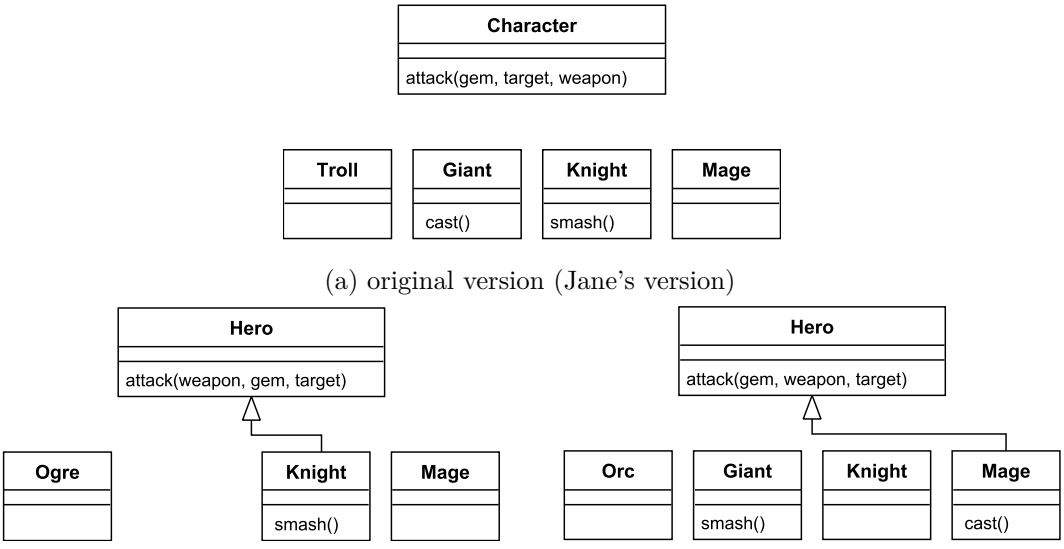


Figure 4.1: An excerpt of the UML-like meta-model of the running example in Figure 4.2.



(a) original version (Jane's version)

(b) left version (Bob's version)

(c) right version (Alice's version)

Figure 4.2: Three incomplete class diagrams of a Role Playing Game.

In this scenario, Jane has set up an initial model of a Role Playing Game (RPG) (Figure 4.2a). She then assigned this work to Bob and Alice. Both Alice and Bob continued to work on the model and made some modifications, seen in Figures 4.2b and 4.2c respectively. Persisting these models in the standard XMI [21] format produces three files as shown in Listings 4.1, 4.2, and 4.3. In this running example, every element has its globally unique ID. Thus, if Bob and Alice create two elements independently, they will not be allocated the same ID. For example, the generalisations that Bob and Alice added in Listings 4.2 and 4.3 have different IDs, leftGen and rightGen respectively.

Listing 4.1: Simplified XMI file of the original version in Figure 4.2a.

```
<uml:Model>
  <packagedElement type=Class id="character" name="Character">
    <operation id="attack" name="attack">
      <parameter id="gem" name="gem"/>
      <parameter id="target" name="target"/>
      <parameter id="weapon" name="weapon"/>
   </operation>
  </packagedElement>
  <packagedElement type=Class id="troll" name="Troll"/>
  <packagedElement type=Class id="giant" name="Giant">
    <operation id="cast" name="cast"/>
  </packagedElement>
  <packagedElement type=Class id="knight" name="Knight">
    <operation id="smash" name="smash"/>
  </packagedElement>
  <packagedElement type=Class id="mage" name="Mage"/>
</uml:Model>
```

Listing 4.2: Simplified XMI file of the left version in Figure 4.2b.

```
<uml:Model>
  <packagedElement type=Class id="character" name="Hero">
    <operation id="attack" name="attack">
      <parameter id="weapon" name="weapon"/>
      <parameter id="gem" name="gem"/>
      <parameter id="target" name="target"/>
    </operation>
  </packagedElement>
  <packagedElement type=Class id="troll" name="Ogre"/>
  <packagedElement type=Class id="knight" name="Knight">
    <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 4.3: Simplified XMI file of the right version of Figure 4.2c.

```
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>
```

An alternative way to persist these three models would be to persist the sequence of all changes through which they were constructed, not to persist their state. This approach was first introduced in [12], and it is illustrated in the next section. This example is extended in Section 7.2 to facilitate explaining the change-based model differencing proposed in this research.

4.3 Proposed Approach

To illustrate the proposed approach, Listing 4.2 shows a state-based representation of Bob's model in Figure 4.2b in (simplified) XMI, and Listing 4.4 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 4.4 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 session in the case of this model. The session at line 1 marks the editing made by

Jane until line 29. Replaying these changes produces Jane's model in Figure 4.2a. The rest of the change events are the modification performed by Bob on Jane's model. Replaying all the changes, both Jane's and Bob's changes, produces the same state as the one captured in Listing 4.2 or Figure 4.2b. Thus, we can conclude that the proposed change-based representation carries at least as much information as the state-based representation.

Such a representation is particularly suitable to identify the changes of the model since the last version. For example, if we can identify that changes recorded for the previous version came before editing session Bob-01 (lines 1–29) of the model, we can readily identify the changes that were 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 4.4 is a simplified version. The real format is in XML-like-format (Appendix A.1). 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>.

Change events that have been persisted to a change-based persistence file cannot be altered or removed. They are immutable. Only new change events can be appended to the file.

4.4 Prototype Implementation

A prototype [61] of the change-based model persistence format (EMF CBP) has been implemented using the model-element level change notification facilities provided by the Eclipse Modeling Framework. In that 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.

Listing 4.4: The complete change events of Bob's model in Figure 4.2b.

```
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"
```

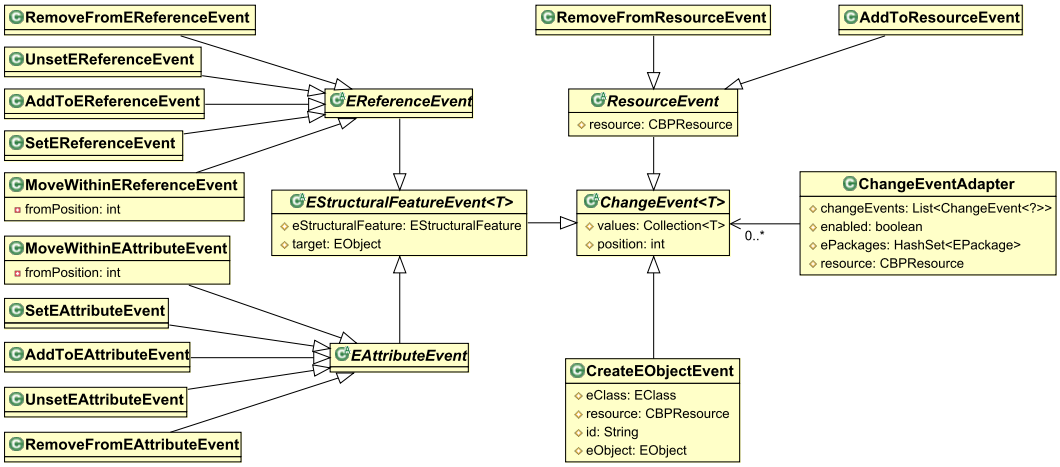


Figure 4.3: Event classes to represent changes of models.

Since not all change events are relevant to change-based persistence (e.g. EMF also produces change notifications when listeners/adapted are added/removed from the model), we have defined a set of event classes to represent events of interest. The event classes are depicted in Figure 4.3 as subclasses of the ChangeEvent abstract class.

EMF has dedicated classes to express the graph structure of a model. For instance, EStructuralFeature can be EReference or EAttribute, it can have a single value or multiple values (e.g., Integer, String), the value(s) of EStructuralFeature can be a EObject or primitive, the EReference can be a containment or non-containment. These characteristics drive the design of the prototype to have different subclasses of ChangeEvent, and they also decide which attributes and methods should be defined in the 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 globally unique ID is assigned to the

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

Listing 4.5: 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 4.5 shows how the prototype handles Notification.UNSET events, based on the type of the feature that was changed. That is, 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 in 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 a 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. To save a model, 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. To load a model, CBPXMLResourceImpl de-serialises every line in the document as a change event and then re-executes it to reconstruct the model. The prototype also includes a CBPXMLResourceFactory class that extends EMF's ResourceFactoryImpl as the factory class for change-based models. Figure 4.4 shows the relationships between these classes.

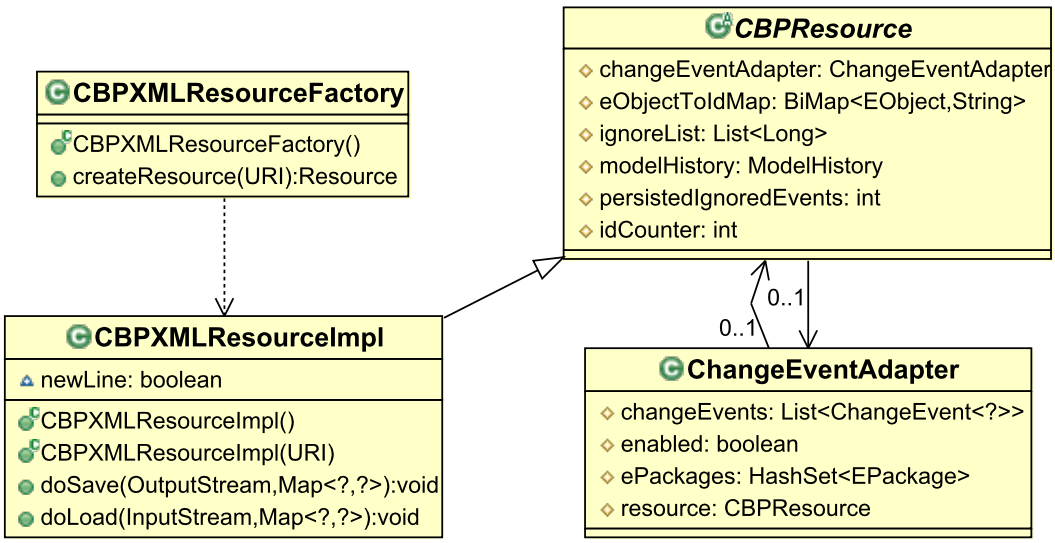


Figure 4.4: Factory, resource, and ChangeEventAdapter classes.

Listing 4.6 shows how to use the prototype in 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 specifies its file as helloworld.cbpxml using URI. The code then executes method startNewSession of cbpResource. This method adds a change event to indicate the start of the editing session as depicted at line 1 in Listings 7.1 and 7.2. The code then uses UMLFactory to create an element, model, of UML2's Model. The code adds element model into cbpResource and sets the name to 'Hello World'. The code then saves the model in change-based format using method save and then unloads cbpResource. Lines 9–12 demonstrate how to replay (load) the model that had been saved and then print the name of the first element in cbpResource, which is expected to print "Hello World".

Listing 4.6: An example how to use CBPResource in Java code.

```
1  /* initialise, save, and unload */
2  CBPResource = (CBPResource) (new CBPXMLResourceFactory()).createResource(URI.
       createFileURI("helloworld.cbpxml"));
3  cbpResource.startNewSession("Initial");
4  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"
```

4.5 Challenges

This section highlights the challenges that come from adopting change-based persistence. As was mentioned in the literature review, change-based persistence also comes with a number of challenges, such as (1) loading overhead and (2) fast-growing model files, which can hold back the delivery of its potential benefits. Addressing these challenges surely facilitates its adoption.

For the first challenge, persisting changes to large models is expected to be much faster and resource-efficient than state-based approaches, since loading models into memory by naïvely replaying the entire change history is expected to have a significant overhead. This work has addressed this challenge by proposing two solutions that reduce the cost of change-based model loading. The first solution is to record and ignore events that are later overridden or cancelled out by other events. That solution can be found in Chapter 5. The second solution is a proposed hybrid model persistence format that uses change-based and state-based persistence together. In that solution, changes applied to a model are persisted into both change-based and state-based representations, but the model is loaded from the stated-based persistence. In that way, it avoids replaying the change events. This solution is discussed in Chapter 6.

In the second challenge—fast-growing model files—persisting a model in a change-based format means that the size of its file grows significantly faster during the model's evolution than it does in its state-based counterpart. This challenge has not been addressed in this research, and must be considered in future work. Nevertheless, this research recommends two solutions. Use sound change-compression operations (e.g. remove older/unused information) to reduce the size of a model in a controlled way, or develop a compact textual format that will minimise the space required to record a change. (A textual line-separated format is desirable to maintain compatibility with file-based version control systems.)

4.6 Conclusions

Through persisting models' change history, this research aims to enable high-performance model differencing and conflict detection in collaborative development settings. This study has translated the concept of change-based persistence into an implementation in a modelling framework, which can be used to persist models.

In this chapter, a running example was introduced. This example is used throughout this thesis to explain the solutions proposed in this study. A prototype of a change-based persistence format also was presented, including its requirements and a design of the implementation that meets the requirements. Some potential benefits and novel capabilities that a change-based persistence can contribute and the challenges that might restrain delivering them also have been presented.

This chapter also has partially addressed the first research question of this study, **How can models be persisted in a change-based format, and how does change-based persistence perform, compared to state-based persistence, in terms of loading and saving models?** (RQ1). To persist models in a change-based format, a prototype has been developed. It captures relevant notifications returned by the notification facilities provided by EMF every time a change is applied to an EMF model. It then transforms the notifications into different classes of change events representing different types of changes (e.g., set, unset, add, remove, move, create, and delete) that conform to the model and meta-model infrastructure of EMF. Every captured change event is then persisted by appending it into an

XML-like-formatted file when the model is saved. The model can be (re)loaded by de-serialising the file and (re)executing all the persisted change events—replaying the historical construction of the model.

Chapter 5

Optimised Loading of Change-based Model Persistence

This chapter introduces and evaluates an efficient approach for loading models stored in a change-based format. This work builds on the change-based model persistence format presented in Chapter 3. It also presents an evaluation on the performance of the proposed loading approach and an assessment of its impact on saving change-based models. The results show that the proposed approach significantly improves loading times compared to the baseline change-based persistence loading approach, and it has a negligible impact on saving.

5.1 Introduction

Saving a model in change-based persistence typically results in a large, ever-increasing file (see Table 2.2) since every change made to the model (even model element deletions) is appended to the file. This also applies to the implementation of change-based model persistence (CBP) in this work, which uses a text file to simplify saving changes by appending them and reading them into memory. The increasing records of changes also cause the loading time of the model to increase, as the loading process has to reconstruct the model's current state from its history [12]. This

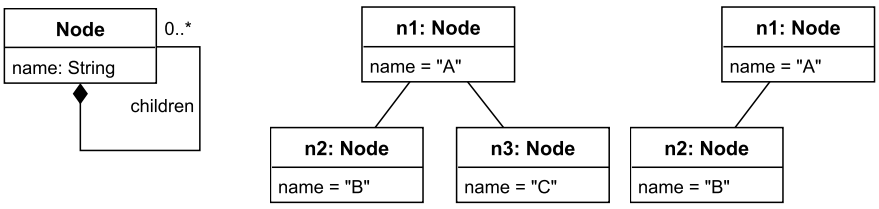
chapter proposes and evaluates an approach that reduces CBP model loading time by avoiding the replaying of historical changes that have no impact on the final state of the model.

The rest of this chapter is structured as follows. Section 5.2 introduces a running example. Section 5.3 presents the proposed approach to speed up model loading and its supporting data structures. Section 5.4 presents experimental results and evaluation. Section 5.5 concludes this chapter.

5.2 Running Example

To explain the optimised loading algorithm for change-based models, this chapter uses the minimal tree meta-model and example tree models in Figure 5.1.

The meta-model for the minimal tree model is expressed in the Eclipse Modeling Framework (EMF) Ecore meta-modelling language, the de-facto standard for object-oriented meta-modelling. The example is contrived to avoid unnecessary repetition, while providing adequate coverage of the core features of Ecore (classes, single/multi-valued features, references). In this example, a tree model consists of named nodes which can—optionally—contain other nodes (child reference).



(a) the tree meta-model (EMF/Ecore) (b) the initial state of a tree model (c) the eventual state after n3 is deleted

Figure 5.1: Running example of a meta-model and a conformant model.

The initial state of the model in Figure 5.1b has three nodes, n1, n2, and n3. It was initially constructed by creating the three nodes (n1, n2, and n3), and nodes n2 and n3 were added as children of n1. The model was modified by deleting node n3 producing the eventual state in Figure 5.1c.

Listing 5.1: State-based representation in simplified XMI of the tree model in Figure 5.1b.

```
1 <Node id="n1" name="A">
2   <children id="n2" name="B"/>
3   <children id="n3" name="C"/>
4 </Node>
```

Listing 5.2: State-based representation in simplified XMI of the tree model in Figure 5.1c.

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

Listing 5.3: Change-based representation of the tree model in Figure 5.1b.

```
1 create n1 of Node
2 set n1.name from null to "A"
3 create n2 of Node
4 set n2.name from null to "B"
5 create n3 of Node
6 set n3.name from null to "C"
7 add n2 to n1.children at 0
8 add n3 to n1.children at 1
```

Listing 5.4: Change-based representation of the tree model in Figure 5.1c.

```
1 create n1 of Node
2 set n1.name from null to "A"
3 create n2 of Node
4 set n2.name from null to "B"
5 create n3 of Node
6 set n3.name from null to "C"
7 add n2 to n1.children at 0
8 add n3 to n1.children at 1
9 remove n3 from n1.children at 1
10 delete n3
```

Listings 5.1 and 5.2 show the simplified XMI format of the models in Figures 5.1b and 5.1c when they are persisted in state-based representation. Listings 5.3 and 5.4 show the change-based representation of the two models respectively, using the CBP syntax introduced in Chapter 4. As both change-based representations show, lines 1–6 record the creation and naming of the three nodes, and lines 7–8 record the addition of n2 and n3 as children of n1. Change-based representation in Listing 5.4 records two additional rows since it also records the recent changes that produce the eventual state of the tree model in Figure 5.1c. Lines 9–10 capture the deletion of n3 (the remove command removes f n3 from its container, and the *delete* command completely removes n3 from its model). Changes in a CBP representation can be uniquely identified by their line numbers.

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

and 10 could be ignored.

5.3 Toward Efficient Loading of Change-Based Models

The flowchart in Figure 5.2 provides an overview of the editing lifecycle of a CBP model [12], with the proposed extensions shown as starred blocks. A model is loaded (1), edited (2), and saved (3). During editing, the changes made to the model are recorded in a memory-based data structure, serialised, and, with the latest events, appended at the end (4). The change events are persisted into a CBP file every time the model is saved (5). When a model is reloaded, the current model state is recreated by replaying the events stored in the CBP file (6).

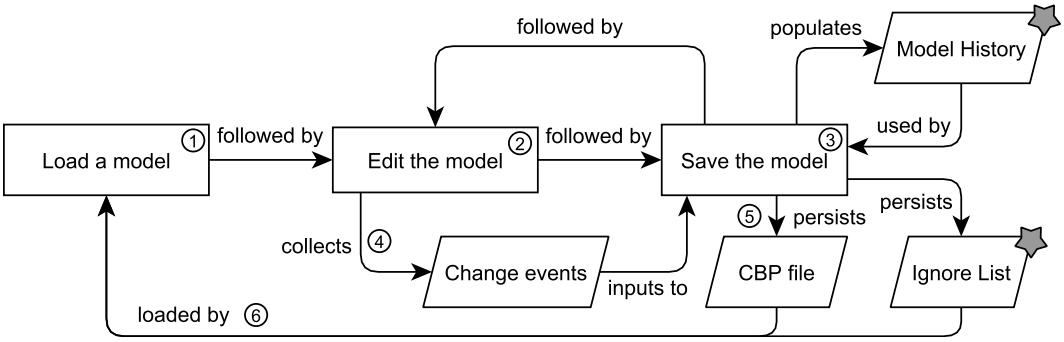


Figure 5.2: CBP workflow, with optimised loading elements indicated by starred blocks.

As mentioned in Section 4.3, the editing history recorded in a CBP file is immutable. As such, superseded events cannot be simply removed from the CBP file. Therefore, the proposed approach adds two artefacts: an in-memory *Model History* data structure, which aggregates change events per model element, and an *Ignore List* file, which persists the position (i.e. line numbers) of superseded events so that the events can be ignored the next time the model is loaded. The Ignore List is saved alongside the CBP file. The rest of this section presents how the Model History is used to detect superseded events and generate the Ignore List.

5.3.1 Model History

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

CBP representation is referred as the *event number*. The proposed data structure is defined in Figure 5.3 using a class diagram.

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

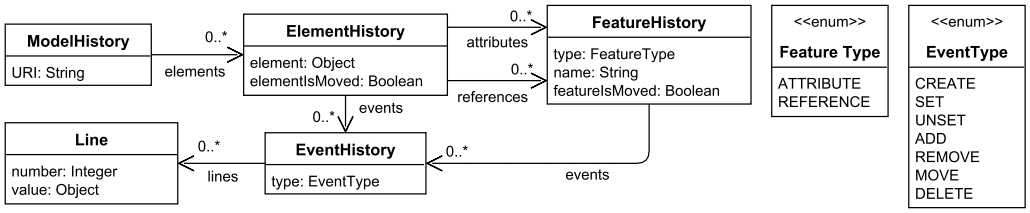


Figure 5.3: The class model defining Model History.

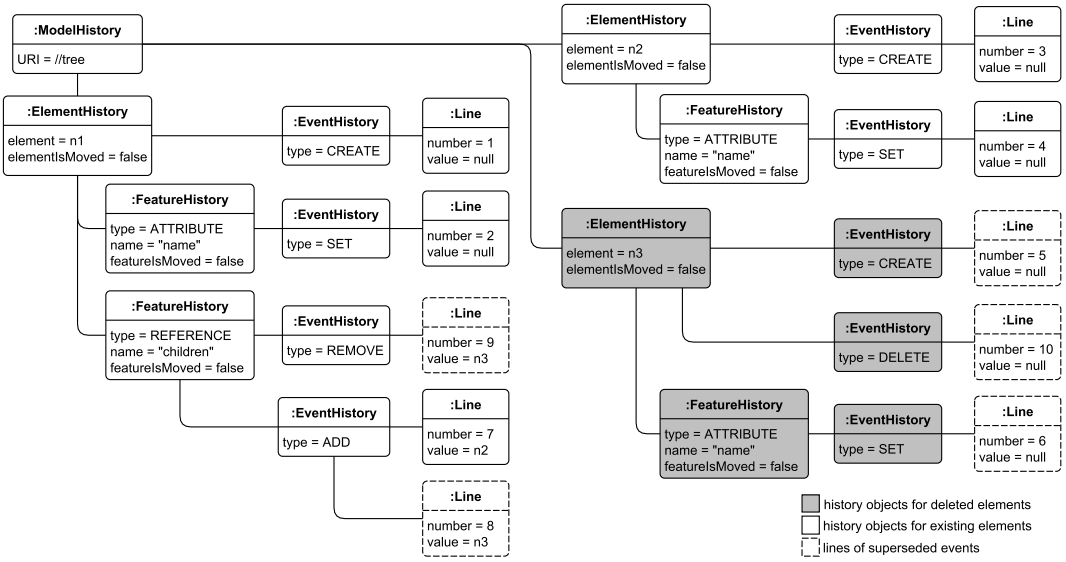


Figure 5.4: The object diagram of the CBP model history in Listing 5.4.

An EventHistory represents a series of events of the same type; it has an attribute type to identify the events' type and can have many Lines. A Line has a number attribute to record the event number and a value that records the element involved in the event (Value is only used for events with types add, remove, and move). Each FeatureHistory can have many EventHistories to represent events that modify the

values of the features. Each ElementHistory can have many EventHistories to represent events that affect the state of the elements (life-cycle and relations to multi-valued features). Figure 5.4 shows an object diagram corresponding to the model in Figure 5.3, which captures the model history shown in Listing 5.4. The grey rectangles are History objects related to the deleted node n3. The rectangles with dashed outlines are Line objects that represent superseded changes.

The following section presents the different strategies used to identify superseded events that will be added to the Ignore List.

5.3.2 Set and Unset Events

During the lifecycle of a model, a single-valued feature can have its value set (assigned) or unset many times. Each event is persisted, but only the last assigned value needs to be considered. For example, in Listing 5.5, the feature name is set to the value "A", unset, and finally set to the value "B". In the final state of the model, n1.name = "B". Thus, only line 4 is significant for the model's final state and therefore lines 2 and 3 can be ignored when loading the model. For a set event, all preceding set and unset events can be ignored, but for an unset event, all set and unset events can be ignored. Executing it does not have any effect on the final state of a model if all the preceding events also have been ignored.

Listing 5.5: A CBP representation of attribute name assignments ended with SET.

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

Listing 5.6: A CBP representation of attribute name assignments ended with UNSET.

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

Based on Listing 5.5, our approach creates an instance of ElementHistory n1, which contains an instance of FeatureHistory name. The FeatureHistory name consists of two EventHistory instances, with types set and unset (the instances are named set and unset respectively for brevity). The set records the *Line* instances that hold the event numbers of the set events, and similarly for unset.

From Listing 5.5, we can thus infer that name.set.lines $= \{2, 4\}$ and name.unset.

lines = {3}. The event numbers in both lists are used to determine that the events represented by lines 2 and 3 are superseded by the event in line 4, which is a set event, giving an ignoreList = {2, 3}. By the same process, for Listing 5.6, we can reason that name.set.lines = {2,3} and name.unset.lines = {4}. However, in this case, the highest-numbered event is an unset, all so line numbers are put into the Ignore List (ignoreList = {2, 3, 4}) (unset event can be ignored along with all preceding set and unset events).

5.3.3 Add, Remove, and Move Events

For a multi-valued feature, add, remove, and move events can be called many times to modify the feature. If an element is added to the feature, moved multiple times, and finally removed, then all the element's preceding events can be ignored, as long as the order of the feature's elements is not changed.

Listing 5.7 shows an example without a move event. In this example, nodes n1, n2, and n3 are added to the children feature of p (lines 5–7). In the latest state of the model, children only contains n1 and n3. As a result, the loading process could ignore the events that represent the add and remove events on n1.

Listing 5.7: A CBP of add and remove operations.

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

Listing 5.8: A CBP representation of add, move, and remove operations.

```
1 create p of Node // children = []
2 create n1 of Node // children = []
3 create n2 of Node // children = []
4 create n3 of Node // children = []
5 add n1 to p.children at 0 // children = [n1]
6 add n2 to p.children at 1 // children = [n1, n2]
```

```
7  add n3 to p.children at 2 // children = [n1, n2, n3]
8  move n1 in p.children from 0 to 1 // children = [n2, n1, n3]
9  remove n2 from p.children at 0 // children = [n1, n3]
```

To create the Ignore List for Listing 5.7, we can deduce that children.add.lines = {{5, n1}, {6, n2}, {7, n3}} (5 is the line number and n1 is the value) and children.remove.lines = {{8, n1}}. Since n2 is removed from its containing feature (line 8), then executing its preceding add and remove events is unnecessary. Note that we retain the create event (line 3) as n2 has not been deleted from the model—only removed from its containing feature. We can iterate through the add and move structures to identify the events on n2 that should be removed, resulting in the ignoreList = {6, 8}.

Listing 5.8 shows an example with a move event[1]. Let's say that a move event is inserted at line 8 (this insertion shifts the remove event of n2 from line 8 to line 9). With the introduction of this move event, we now have the children.add.lines = {{5, n1}, {6, n2}, {7, n3}}, children.move.lines = {{8, n1}}, and children.remove.lines = {{9, n2}}. In the final state of the model, children should have n1 and n3 in order, children = [n1, n3].

However, executing the previous strategy naïvely leads to an erroneous final state. Using ignoreList = {6, 8} produced by the naïve strategy leads to a different order of n1 and n3 in the final state of the model where children = [n3, n1] as shown by the naïve optimised CBP in Listing 5.9. To overcome this problem, $*IsMoved$ flags in Figure 5.3 are used to sign features and elements. If they have been moved—the flags are set to *true*. If an element's $*IsMoved$ flag is true, then all of its line numbers related to add, move, remove events cannot be put into the ignoreList. The flags are set to $false$ if the feature is empty.

Listing 5.9: A naïve optimised CBP representation of original CBP representation in Listing 5.8

```
1  create p of Node // children = []
2  create n1 of Node // children = []
3  create n2 of Node // children = []
4  create n3 of Node // children = []
```

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

```
5  add n1 to p.children at 0 // children = [n1]
6  add n3 to p.children at 1 // children = [n1, n3]
7  move n1 in p.children from 0 to 1 // children = [n3, n1]
```

5.3.4 Create and Delete Events

When an element is deleted, it is completely removed from the model. Therefore, all previous events (*create*, set, unset, move, add, remove, *delete*) on features of the element can be ignored. For example, when node n3 in Listing 5.4 is deleted, the events in lines 5–6 and 8–10 are superseded. If Listing 5.4 is optimised—some of its events are ignored—when loading, it runs as if Listing 5.10 is executed.

Listing 5.10: Change-based representation of the model in Figure 5.1 after removal of node n3.

```
1  create n1 of Node
2  set n1.name from null to "A"
3  create n2 of Node
4  set n2.name from null to "B"
5  add n2 to n1.children at index 0
```

Using Listing 5.4, we can construct the structure of histories that are related to element n3 as follows: n3.*create*.lines = {5}, n3.name.set.lines = {6}, n1.children.add.lines = {{7, n2}, {8, n3}}, n1.children.remove.lines = {{9, n3}}, and n3.*delete*.lines = {10}. Thus, when element n3 is deleted, by iterating through all these history structures, all line numbers associated with n3 can be identified and added to ignoreList producing ignoreList = {5, 6, 8, 9, 10} so they can be ignored in the next model loading.

5.4 Evaluation

This work has developed the proposed efficient loading approach on top of the original CBP implementation [12, 61] and evaluated the approach's model loading performance, its memory footprint, and its impact on the time required to save changes made to CBP models. The evaluation was performed on Intel® Core™ i7-6500U CPU@2.50 GHz 2.59 GHz, 12 GB RAM, and the Java™ SE Runtime Environment (build 1.8.0_162-b12).

Given that CBP is a very recent contribution and we are not aware of any existing

datasets containing real-world models expressed in a change-based format, this work has used synthetic change-based models for the experiments. The synthetic models were derived from real-world data sources: the BPMN2 [62, 63] and Epsilon [64, 65] software projects and the article on the United States [66] in Wikipedia (the article is further referred to as Wikipedia). For the first two projects, for each version of the cases, MoDisco [67] was used to generate a UML2 [68] model that reflects its source code. For the Wikipedia article, a model that conforms to the Modisco XML meta-model [69] was generated. Since these cases have many versions—represented by commits/revisions—different models of the versions can be generated, and to some degree, they reflect the time-ordered changes of the cases. The synthetic change-based model for each case was derived by comparing an initially empty running model to different versions of the case's models sequentially. All identified differences were then reconciled by performing a unidirectional merging to the running model. All changes made to the running model during the merging process were captured and persisted into a CBP file. EMF Compare was used [70] to perform the comparison and merging.

Using the synthetic models, an evaluation was conducted on loading time, saving time, and memory footprint for both loading and saving. To compare the loading time, we ran the optimised and original (baseline) CBP algorithms to reconstruct the current state of each of the three models. (The results are shown in Figure 5.5). As discussed in Section 5.3, optimised CBP also does extra work when saving the changes to a model, in order to save time (relative to original CBP) when loading a model. To analyse the performance of optimisation activities, we compared the overall time required to save a new version of the models described above after one change was made. (The results are shown in Figure 5.6.) This work also compared the memory footprints for both loading and saving, since the optimised CBP approach also requires the maintenance of an additional in-memory data structure that keeps track of element and feature editing histories. (See Figures 5.7 and 5.8 for the results).

For each combination of dimensions (loading time, saving time, loading memory footprint, saving memory footprint), persistence types (original CBP, optimised CBP, and XMI), and cases (BPMN2, Epsilon, and Wikipedia), we conducted measurements

22 times. The results of these measurements enabled us to perform the Welch's t-test [71] to find the significance of the comparisons for each case. This evaluation used a significance level of 5%. If t-test's $p\text{-}value < 0.05$, the null hypothesis (the *means* of the compared persistence types are equal (H_0)) is rejected and the alternative hypothesis (the *means* of the compared persistence types are not equal (H_1), is accepted.

For loading and saving time, this work measured the delta time required for loading and saving. For memory footprint, this work measured the delta of memory used before and after loading and saving. The results are presented below.

5.4.1 Data Description

Table 5.1: Description of change-based models generated for evaluation.

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

Table 5.1 summarises events, elements, and saved versions for the Epsilon, BPMN2, and Wikipedia cases. *Total Events* is the numbers of events that were produced by our approach in generating a change-based model for each case. *Ignored Events* is the number of superseded events that do not need to be replayed when reloading the models. *Elements* is the number of elements contained in each model. *Total Versions* is the number of commits/revisions made to the cases, taken from the Git repositories or from Wikipedia at the time this evaluation was performed. *Processed Versions* is the number of commits/revisions that were processed to produce change-based models: since the comparison between versions takes considerable time, not all versions are processed here.

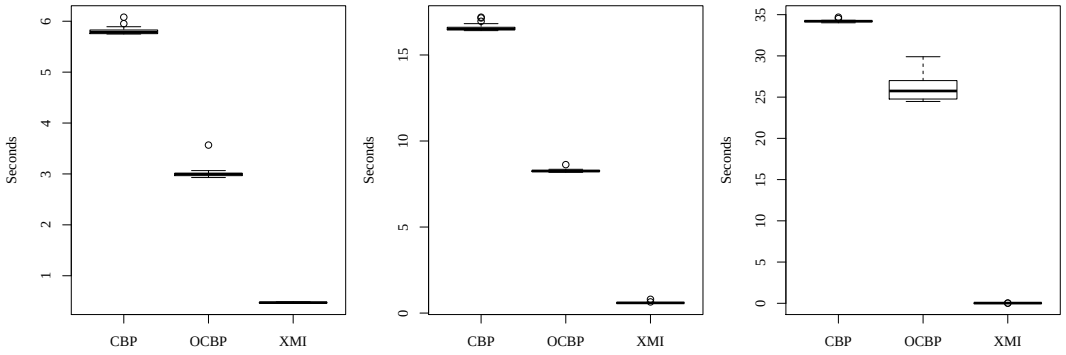
5.4.2 Model Loading Time

This section presents the results of the loading time measurement of change-based models for each pair of persistence types and cases and the t-test results of their comparisons (Table 5.2 and Figure 5.5).

Table 5.2: The t-test results of loading time by original CBP (CBP), optimised CBP (OCBP), and XMI.

Group	Mean	SD	Comparison	t	df	p-value
BPMN2 Load Time (*s*)			**BPMN2 Load Time**			
CBP	5.81	0.08	CBP vs. XMI	315.95	21.46	< 0.05
OCBP	3.02	0.13	CBP vs. OCBP	87.67	35.10	< 0.05
XMI	0.47	0.47	OCBP vs. XMI	93.86	21.18	< 0.05
Epsilon Load Time (*s*)			**Epsilon Load Time**			
CBP	16.60	0.23	CBP vs. XMI	324.18	22.78	< 0.05
OCBP	8.28	0.09	CBP vs. OCBP	160.06	27.48	< 0.05
XMI	0.60	0.05	OCBP vs. XMI	354.52	42.06	< 0.05
Wiki Load Time (*s*)			**Wikipedia Load Time**			
CBP	34.23	0.145	CBP vs. XMI	1,110.10	21.00	< 0.05
OCBP	26.14	1.583	CBP vs. OCBP	23.90	21.35	< 0.05
XMI	0.02	0.001	OCBP vs. XMI	77.37	21.00	< 0.05

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



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

Figure 5.5: Results for loading a model in original CBP (CBP) and optimised CBP (OCBP) and for loading a state-based (XMI) representation.

These loading times show a considerable time saving for optimised CBP: BPMN2 was 48.02% faster, Epsilon 50.12% faster, and the Wikipedia page 23.63% faster than in the original CBP implementation. (All optimised CBP's *means* are smaller than

all original CBP's *means*.) This has a positive correlation to the number of ignored events. All the t-test results also show that loading times for all the persistence types are significantly different (all the *p-values* < 0.05).

For reference, this work also compared CBP loading with the time to load the equivalent state-based model in XMI. Figure 5.5 shows that, even with the improvements delivered by the new algorithm, loading change-based models is still significantly slower than loading a state-based model. (All the XMI's means are smaller than other persistence types' means.)

5.4.3 Model Saving Time

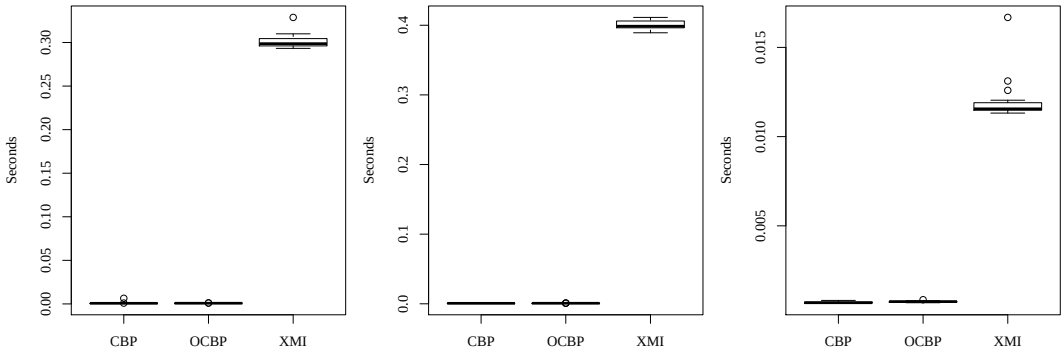
This subsection presents the results of the saving time measurement of change-based models for each pair of persistence types and casez and the t-test results of their comparisons (Table 5.3 and Figure 5.6). As discussed in [12], CBP loading time penalties are balanced against the benefits of CBP in terms of persisting changes (saving time).

Table 5.3: The t-test results of saving time by original CBP (CBP), optimised CBP (OCBP), and XMI.

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

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

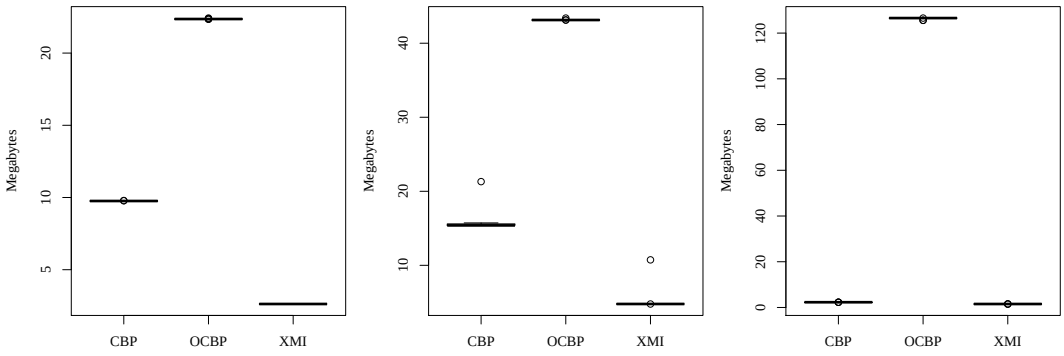
As shown in Table 5.3 and Figure 5.6, the performance of the two CBP implementations is not very different. Since the significance level is 5%, only the BPMN2



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

Figure 5.6: A comparison of the time required to persist an event between original CBP (CBP), optimised CBP (OCBP), and XMI.

case fails. However, the difference between the *means* of its original CBP (0.97 ms) and optimised CBP (0.81 ms) is small. This indicates that the cost of the extra work in the optimised CBP algorithm is negligible. On the other hand, both CBP implementations are significantly faster at saving changes than state-based XMI. (The *means* of both CBP implementations are smaller than XMI's *means*, and both CBP implementations have *p-values* < 0.05 when compared to XMI.) This is expected, as the CBP implementations only need to append the last changes to the existing model file (their performance is thus relative to the number of changes since the last save), while the XMI implementation needs to reconstruct an XML document for the entire state of the model, and it must replace the contents of the model file every time (hence its performance is relative to the size of the entire model).



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

Figure 5.7: A comparison of the memory footprint after loading a model by original CBP (CBP), optimised CBP (OCBP), and XMI.

Table 5.4: The t-test results of the memory footprint after loading a model by original CBP (CBP), optimised CBP (OCBP), and XMI.

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

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

5.4.4 Memory Footprint

The memory footprint after loading models from the three cases is presented in Table 5.4 and Figure 5.7, and the memory footprint after persisting single changes is displayed in Table 5.5 and Figure 5.8. The results show the significant memory overhead of the extra data structure when loading models. (All the *means* of optimised CBP are greater than all the *means* of original CBP and all comparisons between both CBPs show *p-values* < 0.05, Table 5.4.) Both CBPs are also outperformed by XMI in terms of memory footprint when loading models. (All the *means* of XMI are smaller than all the *means* of both CBPs and all comparisons against XMIs show all *p-values* < 0.05, Table 5.4.) In loading, XMI uses significantly less memory than the optimised CBP representation, and it performs slightly better than the original CBP.

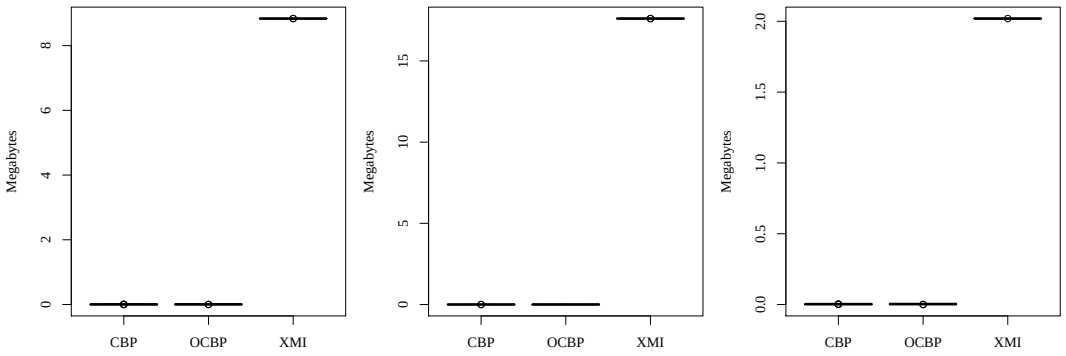
In terms of saving, both CBP implementations persist a single change faster than XMI. (Their *means* are smaller than the *means* of XMI, and all the CBPs' t-tests with XMI show that their differences are significant at *p-value* < 0.05 (Table 5.5.) The optimised CBP has a larger memory footprint than the original CBP. (The means of the optimised CBP for all cases are greater than the means of the original CBP.) However, their memory footprints are not very different. Even though the

BPMN2 and Epsilon cases have *p-values* < 0.05, the differences of the *means* of their original and optimised CBPs are small, and the Wikipedia case also shows *p-value* > 0.05 on its original CBP, compared with the optimised CBP.

Table 5.5: The t-test results of the memory footprint from saving an event by original CBP (CBP), optimised CBP (OCBP), and XMI.

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

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



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

Figure 5.8: A comparison of the memory footprint after persisting an event by CBP, optimised CBP, and XMI.

5.4.5 Discussion

For the original CBP loading, the total time required to load a model is $T_{CBP} = T_E + T_O$, where T_E is the total time required to execute all events, and T_O is the total time needed to complete other required routines (e.g. initialisation, reading files).

For the optimised CBP, the total time to load a change-based model is reduced by the time saved-up by ignoring superseded events T_I, that is $T_{OCBP} = T_E + T_O - T_I$. Thus, it is expected that optimised CBP can load a model faster than original CBP. This statement is in accordance with our finding in Section 5.4.2 that the saved loading time corresponds to the number of ignored events. However, more investigation is required to determine the degree of their correlation, which will be addressed in our future work.

5.5 Conclusions

Change-based persistence can be slow when it comes to loading a model since its change records must be replayed. This study has optimised the loading of change-based persistence by replaying only the change events that affect the eventual state of a model. In other words, the replay ignores change events that are superseded by later change events.

This chapter has proposed an efficient algorithm and supporting data structures for the proposed optimisation. Performance is evaluated on synthesised models, with comparisons to the unoptimised change-based implementation and state-based XMI. Compared to the naïve change-based representation, the optimised version shows considerable savings in terms of loading time with a negligible impact on saving time, but at the cost of a higher memory footprint. However, in terms of loading time and memory footprint, XMI outperforms both approaches but is much less efficient in saving changes.

This chapter has partially addressed the first research question of this study, **How can models be persisted in a change-based format, and how does change-based persistence perform, compared to state-based persistence, in terms of loading and saving models?** (RQ1). Based on the evaluation results, we can state that the performance of change-based persistence on loading models is poor compared state-based persistence. Even though it has been optimised by ignoring replaying change events that are superseded by subsequent change events, it is still significantly outperformed by loading models from their state-based persistence. It also suffers greatly on memory footprint because of the dedicated data structure

employed to track change events. In terms of saving, change-based persistence shows more favourable results than state-based persistence since we need to persist only the recent changes applied to a model rather than saving the entire model. This condition is very favourable when we work with large models in a mature stage where mostly small changes occur.

Chapter 6

Hybrid Model Persistence

Reconstructing a change-based model by replaying its editing history each time the model is queried or modified can get increasingly expensive as the model grows in size. In Chapter 5, we proposed a method to speed up the reconstruction by not replaying change events that do not have any effect on the eventual state of a model. However, that method is still substantially outperformed by loading a model directly from its state-based persistence. In this chapter, we report on a novel approach that integrates change-based and state-based model persistence mechanisms. This hybrid model persistence approach delivers the best of both worlds. This chapter presents the design of the hybrid model persistence approach and reports on its impact on time and memory footprint for model loading, saving, and storage.

6.1 Introduction

Saving models in change-based persistence (CBP) comes at the cost of ever-larger files [2, 10] since all changes (even deleting model elements) are recorded in an editing log, which naturally leads to longer loading times [5]. In Chapter 5, we proposed a method to speed up reconstruction by not replaying change events that do not have an effect on the eventual state of a model. However, the method is still substantially out-performed by loading a model directly from its state-based persistence. Thus, this chapter proposes another solution to address that issue by introducing the concept of hybrid persistence of models. In hybrid model persistence, change-based

representation is augmented with a state-based representation (which can be fully derived from the change-based representation) of the latest state of the model. This is then used to speed up model loading and querying.

This Chapter is structured as follows. Section 6.2 introduces the concept of change-based model persistence and recent work on state-based model persistence. Sections 6.3 and 6.4 present the proposed approach to hybrid model persistence and its implementation. Sections 6.5 and 6.6 present and discuss experimental results and evaluation. Section 6.7 concludes this Chapter.

6.2 Comparing Change- and State-based Model Persistence

Table 6.1 summarises the benefits (+) and drawbacks (-) of change and state-based model persistence. To load a state-based model, only the elements that exist in the final state need to be loaded into memory. To load a change-based persistence model, all the events that lead to the final state must be replayed to load the model in memory. Loading times for state-based models are proportional to the size of the model. Loading times for change-based models are proportional to the number of events. As a result, loading times of change-based models will always increase over time and are considerably longer than for state-based model persistence [5, 13].

Table 6.1: Comparison of model persistence approaches.

Dimension	**Change-based**	**State-based**
Load Time	−	+
Save Time	+	−
Storage	−	+

To store a state-based model, all the elements that exist in the final state must be persisted. To save a change-based model, only the change events in the last editing session need to be persisted. Storing times of state-based models are proportional to the size of the model. Storing times of change-based models are proportional to the number of events in a session. As a result, storing times of change-based models can be considerably shorter than for state-based models [13]. Comparing and finding the differences between two versions of a state-based model is expensive [1] ($O(N^2)$ in the

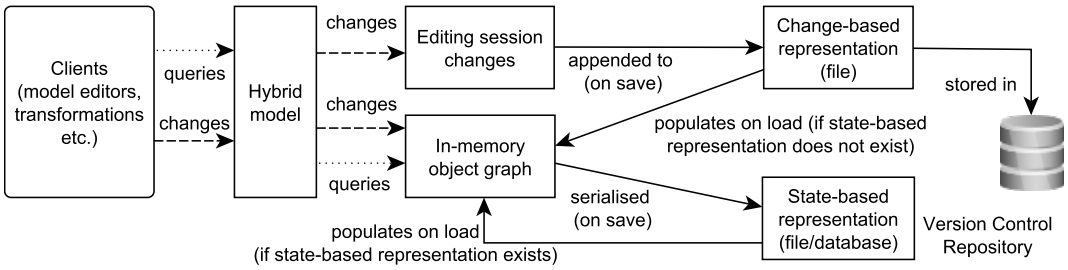


Figure 6.1: The mechanism of hybrid model persistence.

general case) which affects the efficiency of change visualisation and comprehension and has a substantial impact on downstream activities such as incremental model transformation [7] and validation.

The main downsides of change-based model persistence are its model file sizes [2, 10] and ever-increasing loading times [5]. Loading times can be reduced by around 50% by processing the changelog, then detecting, memorising, and subsequently ignoring change events that have no impact on the final state of the model. The loading times are still substantially longer—more than 6.4 times longer and even longer as the persisted changes increase—than loading times for state-based approaches [13].

6.3 Hybrid Model Persistence

To achieve the best of both worlds, this work introduces a hybrid model persistence approach, which combines change-based and state-based model persistence, to work together. An overview of the proposed approach is illustrated in Fig. 6.1. In the proposed approach a *hybrid* model is stored in two representations at the same time: a change-based representation (e.g. using EMF CBP [61]) and a state-based representation (e.g. using XMI [21] or a database-backed approach such as NeoEMF [16]). The change-based representation is treated as the main representation of the model, while the state-based representation can be fully derived from the change-based representation.

Loading a hybrid model. Models are loaded into in-memory object graphs that clients (e.g. editors, transformations) can then interact with. Depending on the state persistence mechanism, the object graph may be loaded in its entirety at startup (e.g. XMI) or loaded progressively, in a lazy manner (e.g. NeoEMF/CDO [16, 29]).

In the proposed hybrid approach, if the state-based counterpart already exists, the in-memory object graph is populated from it. Otherwise, it is populated by replaying the complete editing history recorded in the change-based representation.

Changing a hybrid model. When an element in a loaded model is created, modified, or deleted, the change is applied to the in-memory object graph, and it is also recorded in an in-memory list of changes (*Editing session changes* in Fig 2). This work uses the term *editing session* for the period between loading a model and saving it back to disk.

Saving a hybrid model. The current version of the in-memory object graph is stored in the preferred state-based representation. The list of changes recorded in the current editing session (with optional processing, as described above) is appended to the change-based representation.

Versioning a hybrid model. Since the state-based representation is fully derived from the change-based representation, if a model needs to be versioned (e.g. in a Git repository), only the change-based representation needs to be stored. The first time it is loaded after being checked out/cloned, the state-based representation is computed and persisted locally and is used in subsequent model loading steps.

Comparing hybrid models. To compare two hybrid models—discussed in Chapters 7 and 8, their change-based representations are used. This is much more efficient than state-based comparison.

6.4 Implementation

This work has implemented the proposed hybrid model persistence approach in a prototype [61] on top of the Eclipse Modeling Framework (EMF) [20]. The prototype makes use of an existing implementation of change-based model persistence, the EMF CBP [12], augmented with two state-based model persistence implementations: NeoEMF [16] and XMI [21].

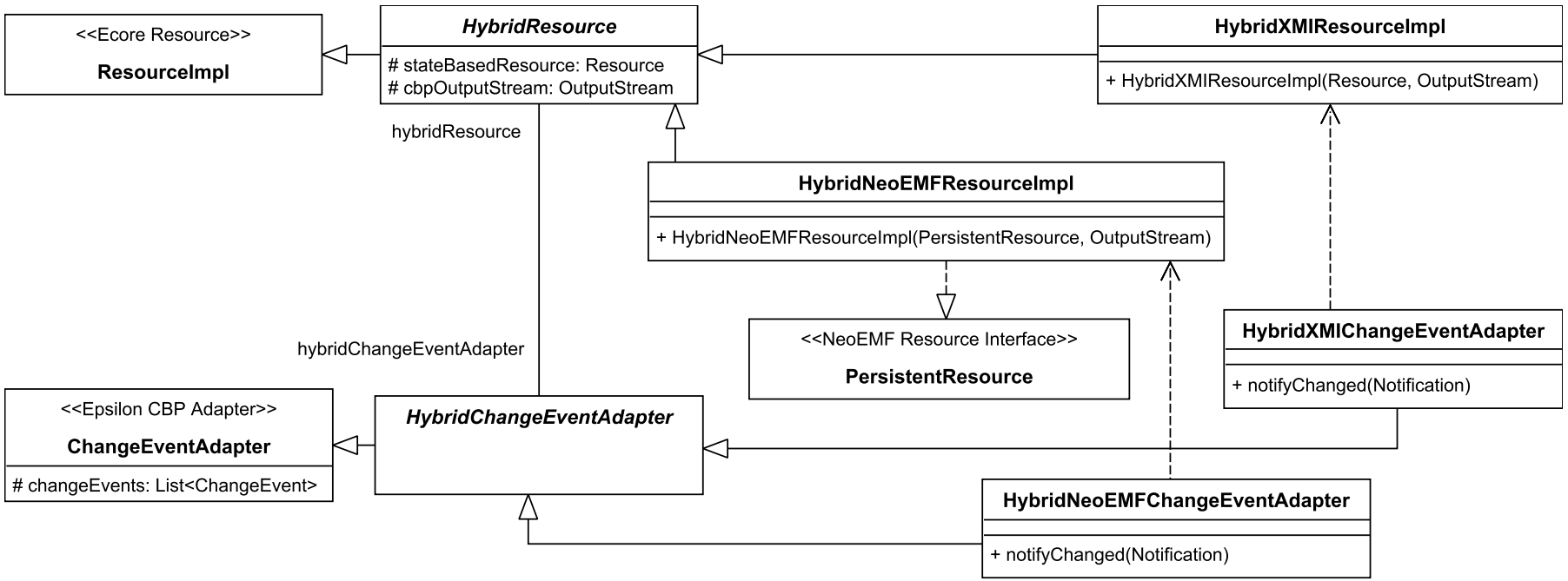


Figure 6.2: Class diagram showing the core components of the hybrid model persistence implementation.

XMI has been selected as a standard state-based model persistence format (natively supported by EMF), and NeoEMF as a best-of-breed representative of database-backed state-based model persistence framework. The core components of the prototype are presented in Fig. 6.2.

The EMF CBP provides a ChangeEventAdapter class [12] that extends from Ecore's EContentAdapter adapter class [72]. This class collects changes made to the in-memory object graph of an EMF model in the form of a list of events, ChangeEvents. Based on this class, this work derived an adapter class, HybridChangeEventAdapter, for the hybrid model persistence implementation. It is an abstract class, so it can be further derived to create different implementations of adapter classes for different types of state-based model persistence. The HybridNeoEMFChangeEventAdapter is the adapter class for NeoEMF; and the HybridXMIChangeEventAdapter, for XMI. These classes override notifyChanged(Notification) in the ChangeEventAdapter class, to handle events that are specific to NeoEMF and XMI, respectively.

This work also created a resource class for hybrid persistence, HybridResource, derived from the Ecore's ResourceImpl [73]. (A resource class is a class dedicated to interacting with a persistence, e.g. save, load, get contents.) This class also is abstract so that it can be realised in different resource implementation classes for different state-based model persistence. The HybridResource class contains the stateBasedResource field, which is used to refer to the state-based model persistence that is being used, and the cbpOutputStream field that refers to an OutputStream (e.g. file, in-memory) as the representation of the change-based model persistence for saving changes. HybridResource has an association with HybridChangeEventAdapater, so that the former can access the events collected by the latter, and the latter can also use facilities provided by the former (e.g. getting the identity of an element in the resource; saving changes to a change-based model representation).

The resource implementation classes for NeoEMF and XMI are HybridNeoEMF-ResourceImpl and HybridXMIResourceImpl, respectively. HybridNeoEMFResourceImpl also implements the NeoEMF's PersistenceResource interface [74], so that specific NeoEMF methods can be used (e.g. close() to close a connection with a backend database).

6.5 Evaluation

In this section, this work compares hybrid model persistence (EMF CBP with NeoEMF and with XMI) vs. state-based model persistence (NeoEMF or XMI only) on storage space usage, loading and saving time, and memory footprint, and it demonstrates that hybrid model persistence can still perform fast model loading and saving.

The evaluation was performed on Intel® Core™ i7-6500U CPU @ 2.50 GHz 2.59 GHz, 12 GB RAM, and the Java™ SE Runtime Environment (build 1.8.0 _162-b12). For the evaluation, this work used models reverse-engineered from the Java source code of the Epsilon [64, 65] and BPMN2 [62] projects. For state-based representation of the models, this work used the MoDisco tool [67] to generate XMI-based UML2 [68] models that reflect the classes, fields, and operation signatures of the source code of the project and then imported the generated models into NeoEMF. This work also derived MoDiscoXML models [69] from the article on the United States in Wikipedia [66]. This work then used reverse-engineering to generate a change-based model persistence for each project, based on the differences between consecutive versions of the models.

6.5.1 Storage Space Usage

For the Epsilon project, this work successfully generated a change-based model persistence from version 1 up to version 940 and also change-based model persistence for the BPMN2 project and the Wikipedia article up to version numbers 192 and 10,187 respectively. The details (element count, event count, space size, and average space size per element or event) of their models, when persisted in XMI, NeoEMF, and EMF CBP are shown in Table 6.2. The last column of the table derives an average space usage per element (for state-based model persistence) or event (for change-based model persistence). Thus, we can estimate the storage space usage for a hybrid model persistence to be the combined space usage of change-based model persistence and the appropriate state-based model persistence.

Table 6.2: Space usage for the Epsilon and BPMN2 projects and the Wikipedia article on the United States (m = million events, MB = Megabytes, KB = Kilobytes).

Case	Generated from	Type	Element Count	Event Count	Space Size	Average Space Size
Epsilon	940 commits	XMI	88,020	—	9.44 MBs	112 bytes/ element
		NeoEMF	88,020	—	188 MBs	2 KBs/ element
		Epsilon CBP	—	4.3 m	406 MBs	98 bytes/ change event
BPMN2	192 commits	XMI	62,062	—	6.55 MBs	110 bytes/ element
		NeoEMF	62,062	—	134 MBs	2 KBs/ element
		Epsilon CBP	—	1.2 m	109 MBs	92 bytes/ change event
Wikipedia	10,187 versions	XMI	13,112	—	1.28 MBs	102 bytes/ element
		NeoEMF	13,112	—	31.8 MBs	2 KBs/ element
		Epsilon CBP	—	62.3 m	5.85 GB	98 bytes/ change event

6.5.2 Time and Memory Footprint of Loading and Saving Models

This work evaluated the performance of our hybrid persistence prototype against XMI and NeoEMF regarding time and memory footprint for loading and saving. In the evaluation, experiments were repeated 22 times for each dimension measured. Since the data were not normally distributed, this work used the nonparametric Mann-Whitney U test [75] with 5% significance level.

As seen in Table 6.3, all cases experience a slight slowdown on loading and saving time (hybrid approach's *mean* > state-based approach's *mean*). However, for almost all NeoEMF cases, the slowdown is not significant. This means that the side-effect of the hybrid approach on loading and saving time is still negligible. The hybrid approach also produces a higher memory footprint than the state-based-only approach.

Table 6.3: A comparison of the time and memory footprint for loading and saving models of the hybrid and state-based-only persistence. Time is in seconds, and the memory footprint is in MB.

Dimension	**Case**	**Backend**	**Hybrid**		**State-based**		**Significance**	
			mean	*sd*	*mean*	*sd*	*W*	*p-value*
Loading Time	Epsilon	NeoEMF	0.292	0.061	0.279	0.023	258	**0.72**
		XMI	0.317	0.006	0.270	0.018	26	< 0.05
	BPMN2	NeoEMF	0.308	0.071	0.286	0.025	230	**0.79**
		XMI	0.212	0.016	0.179	0.016	37	< 0.05
	Wikipedia	NeoEMF	0.262	0.048	0.273	0.062	250	**0.86**
		XMI	0.045	0.001	0.040	0.001	0	< 0.05
Saving Time	Epsilon	NeoEMF	0.0892	0.0421	0.0829	0.0494	216	**0.55**
		XMI	0.411	0.023	0.397	0.015	78	< 0.05
	BPMN2	NeoEMF	0.0777	0.0424	0.0775	0.0452	213	**0.51**
		XMI	0.33	0.007	0.28	008	0	< 0.05
	Wikipedia	NeoEMF	0.135	0.048	0.120	0.024	218	**0.59**
		XMI	0.024	0.048	0.020	0.002	42	< 0.05
Loading Memory Footprint	Epsilon	NeoEMF	38.601	0.878	10.014	1.088	0	< 0.05
		XMI	10.72018	0.00022	10.72009	0.00024	0	< 0.05
	BPMN2	NeoEMF	40.78	1.29	27.20	1.05	0	< 0.05
		XMI	6.73367	1.29305	6.73367	0.00056	101	< 0.05
	Wikipedia	NeoEMF	35.91	1.03	27.25	0.54	27.25	**0.54**
		XMI	8.4079	0.0008	8.0933	0.0009	0	< 0.05
Saving Memory Footprint	Epsilon	NeoEMF	2.64	1.29	2.61	0.78	283	**0.34**
		XMI	1.56355	0.0005	1.56326	0.0018	408	< 0.05
	BPMN2	NeoEMF	1.86	3.86	1.52	0.77	308	**0.12**
		XMI	0.8378	0.00361	0.8375	0.00362	58	< 0.05
	Wikipedia	NeoEMF	1.32	1.51	0.97	0.76	189	**0.22**
		XMI	0.0010	0.00044	0.0005	0.00001	0	< 0.05

6.6 Discussion

The use of state-based model persistence in hybrid model persistence enables loading performance that is comparable to the performance of loading only from a state-based persistence, as shown by the evaluation of loading time in Section 6.5.2. In this way, model loading does not have to replay all the changes persisted in its change-based model persistence—the main challenge for the change-based approach [5, 13]. Hybrid model persistence performs slightly more slowly—statistically significant for Hybrid XMI but insignificant for Hybrid NeoEMF—compared to loading a state-based model. A slight slowdown also appears on model saving—statistically significant for Hybrid XMI but insignificant for Hybrid NeoEMF (Section 6.5.2). The slowdown is caused by persisting changes into two representations.

The main drawback of hybrid model persistence is that it consumes more memory when loading and saving, and it requires more storage space for persisting models than state-based representation only (Sections 6.5.2 and 6.5.1). However, considering the cost of main memory and storage, the trade-off can be acceptable in most real-world scenarios. The summary of the findings are shown in Table 6.4.

Table 6.4: Hybrid model persistence compared to other persistence approaches.

Dimension	Change-based	State-based	Hybrid
Load Time	−	+	+
Save Time	+	−	+
Storage/Memory	−	+	−

6.7 Conclusions

Change-based persistence can be slow when it comes to loading a model, since its change records must be replayed. While Chapter 5 tried to address this by not replaying change events that do not have any effect on eventual states, its performance was not at level to outperform persisting in XMI. So, this study implemented a hybrid model persistence—using change-based and state-based persistence together—where models are loaded from the state-based persistence but changes are saved in both persistences.

This chapter has evaluated the impact of hybrid persistence on time and memory footprint for model loading and saving and for usage of storage space. The evaluation showed that the hybrid model persistence provides benefits on model loading time, since its performance is comparable to loading a model from a change-based persistence only, with trade-offs on increased memory footprint and storage space usage.

This chapter also partially addressed the first research question of this study, **How can models be persisted in a change-based format, and how does change-based persistence perform, compared to state-based persistence, in terms of loading and saving models?** (RQ1). Based on the evaluation, it is best to persist models in hybrid model persistence since it experiences only a slight slowdown on both loading and saving, compared to persisting models in state-based persistence. In other words, the side-effect of the hybrid approach on loading and saving time is negligible. However, it comes with trade-offs of a larger memory footprint and more storage space.

Chapter 7

Efficient Model Differencing of Change-based Models

In Chapters 5 and 6, this work proposed two approaches to optimise the loading of change-based model persistence. This chapter presents a method for using change-based persistence in certain circumstances to identify differences between two versions of a model more efficiently than by using state-based persistence. A detailed discussion of the proposed change-based model differencing and its evaluation also is presented in this chapter.

7.1 Introduction

In modelling and model management, it is common to find that many versions or variants of a model exist. These versions are commonly persisted as snapshots of the model at a given point in time in a state-based format such as XMI. Model differencing activities can be applied to versions of a model to highlight such differences as changes in properties and values, new/deleted elements, etc. However, comparing versions of large file-based models in a state-based format can be computationally expensive, since every element of two versions being compared must be loaded into memory to be matched and diffed.

In previous publications from this research [12–14], change-based model persistence (CBMP) was proposed as an alternative to state-based model persistence of EMF

models [20]. Instead of persisting models as XMI snapshots, models are persisted as a complete history of changes in the proposed approach. We demonstrated the substantial performance benefits of change-based model persistence in terms of saving changes to large models [12], and we proposed a method to reduce model loading time compared to naïvely replaying all recorded change events [14] to reconstruct the state of a change-based model. This chapter demonstrates how a change-based representation also enables much more efficient and performant model differencing between versions of the same model. Our experiments, presented in Section 7.5, demonstrate savings in the order of 90% for (relatively) small changes made to large models.

This chapter is structured as follows. Section 7.2 extends the running example from Section 4.2 to explaining the differencing approach proposed in this chapter. Section 7.3 presents the way that state-based model differencing performed in EMF Compare [30]. Section 7.4 presents our change-based approach to speed up model differencing and its implementation. Section 7.5 reports the results of experiments used to evaluate the proposed approach. Section 7.6 concludes this chapter.

7.2 Running Example: Part II

In this section, we extend the running example presented in Section 4.2. Using the change-based model persistence presented in Chapter 4, instead of persisting the models in Figure 4.2 only in state-based format, we can also persist the complete history of changes of the models in change-based format.

Listing 7.1: Change-based representation of the original version in Figure 4.2a.

```
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 4.2a is persisted in a change-based model representation in Listing 7.1. The change events (Listing 7.2) made by Bob are appended to Jane's original change events. Thus, the change events that represent Bob's version (Figure 4.2b) comprise the original change events and the change events (Listing 7.2) that he made. (Only the appended changes are presented on that list.) The change events that represents Alice's version (Figure 4.2c) are presented in Listing 7.3. One clear advantage of change-based model persistence is that, from Listing 7.2, we can immediately know all the changes made by Bob and Alice (starting from line 30), and we can identify all the elements that have been modified since Jane's version.

Listing 7.2: The appended events made by Bob to produce Figure 4.2b.

```
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 7.3: The appended events made by Alice to produce Figure 4.2c.

```
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 4.2a, 4.2b, and 4.2c as well as Listings 7.1, 7.2, and 7.3 occurred according to the following story.

Jane, as the technical leader, set up the initial model. The events of the initial set-up are recorded in the CBMP in List. 7.1. 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). Finally, she pushed her work to a change-based version control system. If her work is visualised in state-based format, the model looks like Figure 4.2a.

Then Jane assigned work to Bob and Alice. Both of them checked out this project to their own machines. Alice continued the model. She 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 before the rest of the parameters (List. 7.3, line 31). She also moved operation smash from class Knight to class Giant and operation cast from class Giant to class Mage as it is more reasonable that they belong to their new classes (lines 32–35). Alice also created a generalisation relationship with ID rightGen from class Troll to class Character (lines 36–39). Bob did the same thing except that his generalisation came with ID leftGen (List. 7.2, lines 31–33).

Later on, Jane informed Alice and Bob that she wanted all good characters to be derived from a general, hero-like class, and the enemy should be Orcs, not Trolls. She also instructed Bob to focus on developing class Knight and Alice on class Mage. As a result, Alice 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). Unfortunately, Bob deleted class Giant accidentally (lines 38–42). The class diagrams of Bob's and Alice's models are in Figures 4.2b and 4.2c respectively. Finally, Alice changed the name of class Troll to "Orc" (line 42) while Bob changed it to "Ogre" (line 43).

In Listings 7.3 and 7.2, 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 transaction—identified with the same composite ID. For example, moving an element from one 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 in Listing 7.3).

7.3 State-based Model Differencing

Referring to the example in Section 7.2, Bob decides at some point to compare his model to Alice's model because he is interested in analysing the differences between their models. Bob uses a model differencing tool to perform state-based model differencing. In state-based model differencing, comparing models commonly consists of two steps: *matching* and *diffing*. The matching process establishes similarities between the elements of two models, to determine the elements in the left model that correspond to elements in the right model. Generally, the matching process iterates through all the elements of the models being compared and matches them

by their identifiers or through a similarity mechanism [30, 56]. The diffing process then identifies differences between the matched elements [30, 56].

In our example, the matching process in state-based comparison—as performed by EMF Compare [30]—iterates through all the elements of both models and matches them using their identifiers. The matching process yields 10 matches: m_1 = (character, character), m_2 = (attack, attack), m_3 = (gem, gem), m_4 = (weapon, weapon), m_5 = (target, target), m_6 = (troll, troll), m_7 = (knight, knight), m_8 = (smash, smash), and m_9 = (mage, mage), and 3 unmatched elements, um_1 = (-, giant), um_2 = (-, rightGen), m_3 = (-, cast), and um_4 = (leftGen, -).

The diffing process then iterates through all the matches and uses an LCS algorithm to identify the differences. During this iteration of the second match m_2, the algorithm determines that, to make the left feature parameters equal to the right feature parameters, parameter gem must be moved from index 1 to 0 (diff ds_1). It is important to note that the LCS algorithm does not detect the different position of parameter weapon; it only identifies the minimum number of differences which, if all are resolved unidirectionally, can make the two models equal.

In the match m_6, the diffing process determines that the classes troll are different in their name. The left troll's name is "Ogre" while the other troll's name is "Orc" (diff ds_2). In the eighth match m_8, the diffing process determines that the containers of operation smash are different. Thus, element smash must be moved from knight's operations to giant's operations (diff ds_3). For the other matches, the diffing process does not identify any differences.

From the unmatched elements (um_1, um_2, um_3, and um_4), the diffing process determines that, to make the left model equal to the right model, class giant must be added to the left model's resource at index 2 (diff ds_4), generalization rightGen must be added to class mage's generalization (diff ds_5), operation cast must be added to class mage's operations (diff ds_6), and generalization leftGen must be removed from class knight's generalization (diff ds_7).

Differences are commonly expressed as a list of changes that must be applied to a target model to make it equal to a reference model. This work treats the left model as a reference model and the right model as the target model. This means that

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

Based on these definitions, this work can express the result of the diffing process as: $ds_n = [LeftContainer_n, RightContainer_n, LeftFeature_n, RightFeature_n, LeftIndex_n, RightIndex_n, LeftValue_n, RightValue_n, Kind_n]$. Therefore:

ds_1 = [attack, attack, parameters, parameters, 0, 1, gem, gem, MOVE]
ds_2 = [troll, troll, name, name, 0, 0, “Ogre”, “Orc”, CHANGE]
ds_3 = [knight, giant, operations, operations, 0, 0, smash, smash, MOVE]
ds_4 = [resource, resource, null, null, null, null, null, giant, DELETE]
ds_5 = [mage, mage, generalization, generalization, null, 0, null, rightGen, DELETE]
ds_6 = [mage, mage, operations, operations, null, 0, null, cast, DELETE]
ds_7 = [knight, knight, generalization, generalization, 0, null, leftGen, null, ADD]

We use this information to represent the diffs visually in Figure 7.1. We can also transform these diffs into change events that, if the diffs are executed as changes to the right model, they transform it into the left model and generate relevant change events. The change events are presented in Listing 7.4. Diff ds_1 produces the change event at line 1 in Listing 7.4, ds_2 produces line 2, ds_3 produces lines 3–4, ds_4 produces lines 5–7, ds_5 produces lines 8–10, ds_6 produces lines 11–13, and ds_7 produces lines 14–16.

Listing 7.4: Diffs presented as change events.

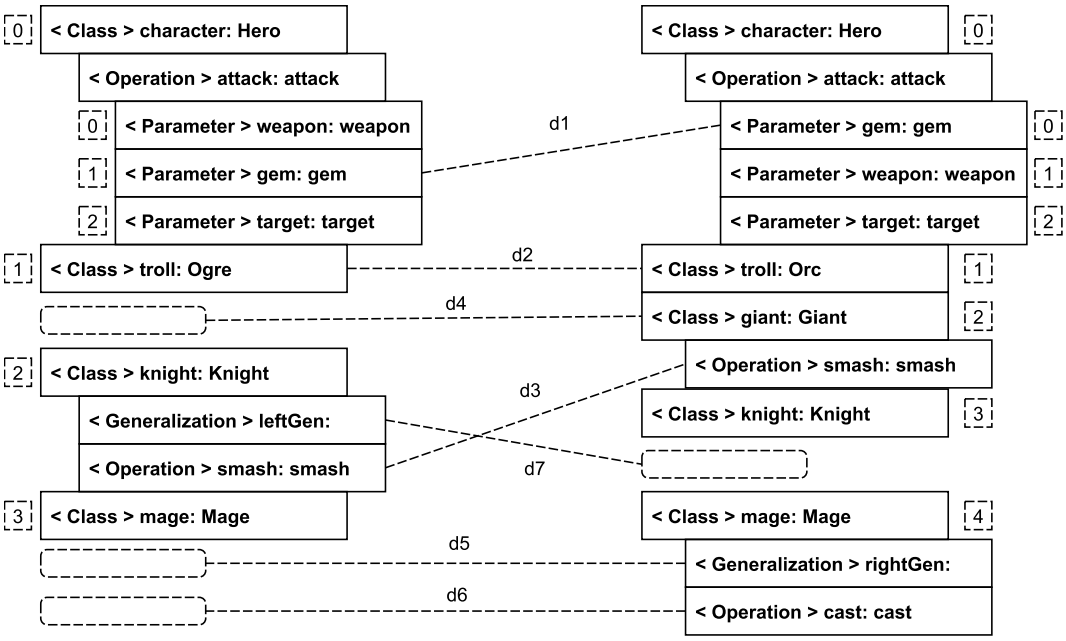


Figure 7.1: A comparison of the left and right models in Listings 4.2 and 4.3.

```
1  move gem in attack.parameters from 0 to 1
2  set troll.name from "Orc" to "Ogre"
3  remove smash from giant.operations at 0 composite c1
4  add smash to knight.operations at 0 composite c1
5  unset giant.name from "Giant" to null composite c2
6  remove giant from resource at 2 composite c2
7  delete giant composite c2
8  unset mage.generalization from rightGen to null composite c3
9  unset rightGen.general from character to null composite c3
10 delete rightGen composite c3
11 unset cast.name from "cast" to null composite c4
12 remove cast from mage.operations composite c4
13 delete cast composite c4
14 create leftGen type Generalization composite c5
15 set knight.generalization from null to leftGen composite c5
16 set leftGen.general from null to character composite c5
```

7.4 Change-based Model Differencing

Compared to the state-based model conflict detection of EMF Compare, the change-based model conflict detection proposed in this work consists of three phases: event loading, element tree construction, and conflict computation. Conflict detection is not performed over all the elements of the model, as it is in state-based model differencing. Instead, this approach needs to compare only the last sets of change

events of the two models, starting where the lines of the two models are different. A simplified class diagram of this approach [61] is depicted in Figure 7.2. The three phases are described in detail in the following sections.

7.4.1 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 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 7.1 are skipped. Only the lines starting with line 30 in Listings 7.2 and 7.3 are loaded. This yields two partial—left and right—change-event models.

7.4.2 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 information similar to that captured in change lists in state-based model persistence, but it also provides more information about the changes. For example, the element tree can keep track of a feature's old value and an element/value's indexes inside multi-valued properties. The element tree contains only the partial states of affected elements of the original, left, and right models as depicted in Figures 7.3 and 7.4.

To better understand the construction of an element tree from change events, we use the following running example using both change events in Listings 7.2 and 7.3. We start from the left change events.

Left Side

In the first change event in Listing 7.2 at line 30, the change event is a session event. It indicates that all the following change events until the final line or next session event are persisted in one batch when they are saved. At line 31, we can see that Bob created a Generalization with ID leftGen. Thus, in elementTree, an element with ID leftGen also is created. To indicate that an element is newly created in the session, we put a '+' sign at the left lower box of element leftGen in Figure 7.3.

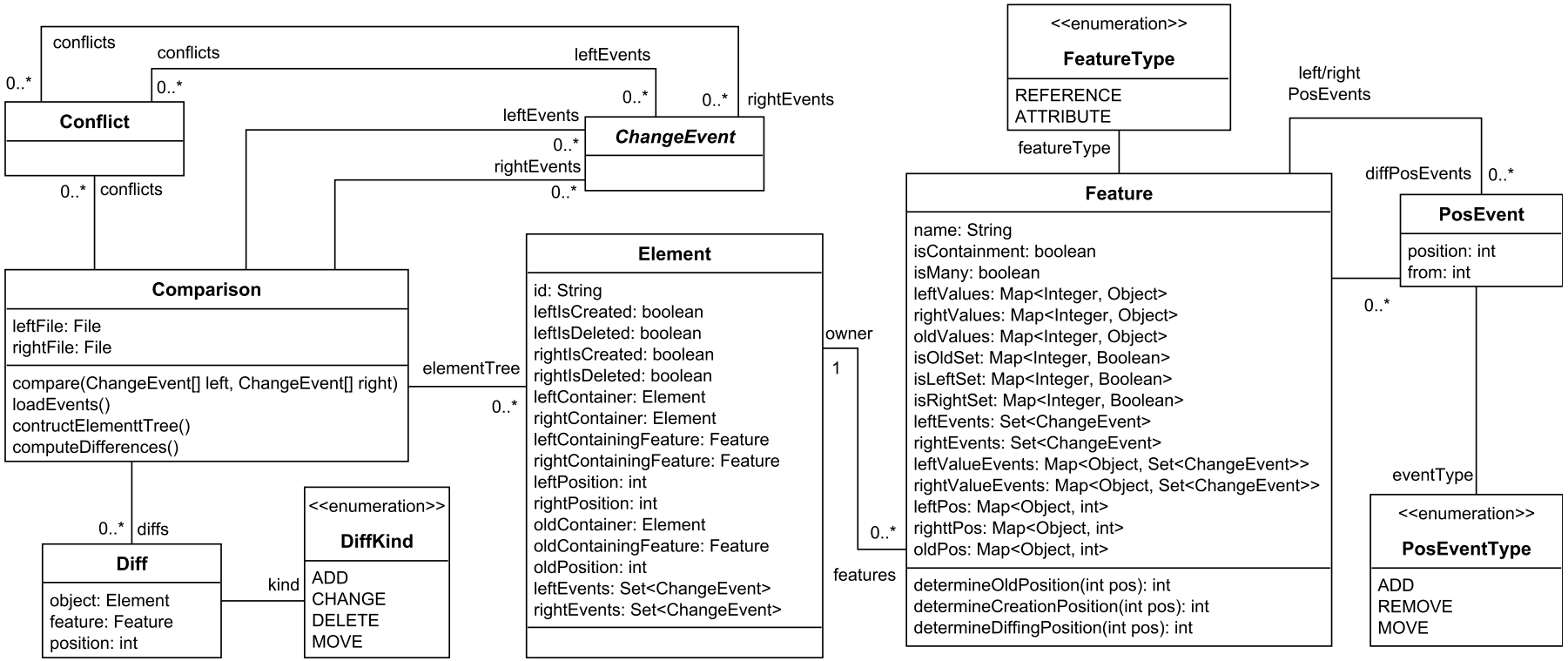


Figure 7.2: A class diagram showing the core components of the change-based approach to speed up model differencing and conflict detection.

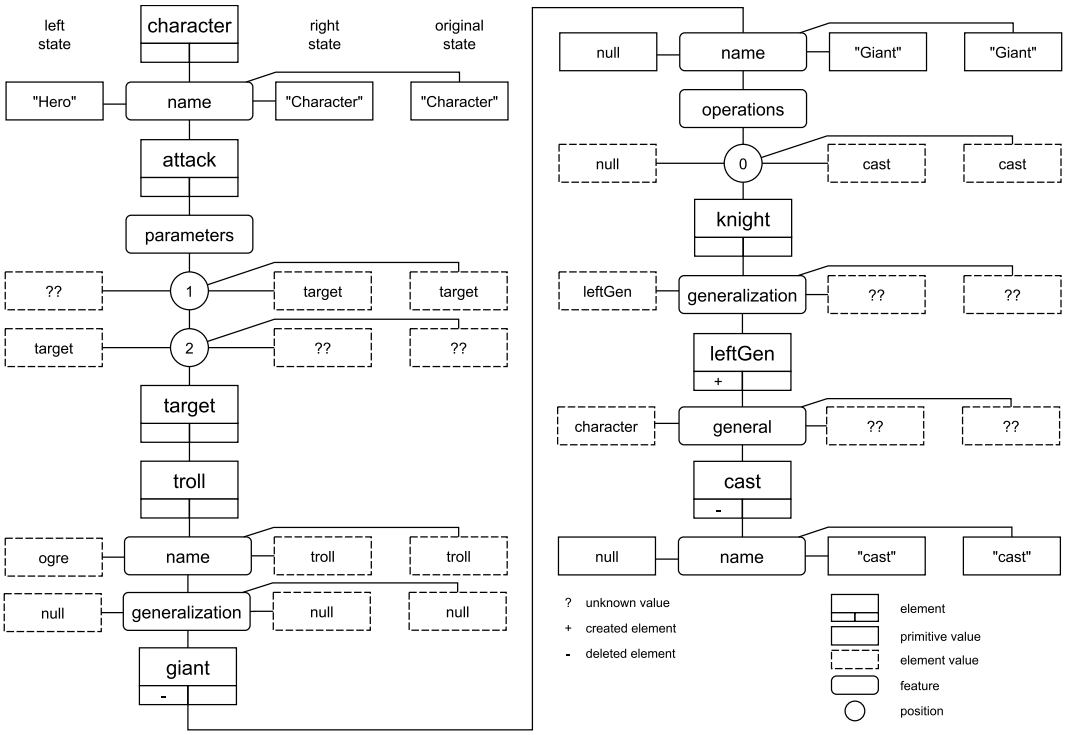


Figure 7.3: An element tree constructed from information in CBPs in Listing 7.2 (left change events only).

At line 32, the feature general of leftGen is set to character. From the change event, we can recognise that character existed in the previous version since it has not been created in the current editing session. Thus, we create an element with ID character and the feature general of leftGen and put them in elementTree. We then set the value of general to character on the left side. We follow the same routine with troll and generalization at line 33, adding element troll and feature generalization to elementTree and setting the value of feature generalization to leftGen on the left side of the elementTree.

The change event at line 34 changes character's name from "Character" to "Hero". From the change event, we can see that character 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 the 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 also is set to "Character", but it will be modified later when we process the right change events (Alice's change events) if there is any change event that affects

it. The same routine is applied when we process the change event at line 43 later.

Lines 35 and 36 are the change events of composite move event l1. Element leftGen is removed (unset) from troll's generalization and is assigned (set) to knight's generalization. From these change events, we can see that element knight also existed in 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 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 see that element target has been 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 when 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 that element and its sub-elements, removes the sub-elements from their containers, and deletes the element and sub-elements from the model. As can be seen, the value of cast's name is unset from "cast" to null at line 38. From the change event, we know that cast has 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 see that giant and its feature operations exist, 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 a '-' sign on the left side to indicate that the element is deleted from the model in the left version.

Change event at line 41 is similar to change event 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 a '-' sign to indicate that the element is deleted from the model in the left version.

Figure 7.3 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.

Right Side

In Listing 7.3, 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 determine the index of target only 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 see that smash is no longer in knight's operations; it is 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. This event

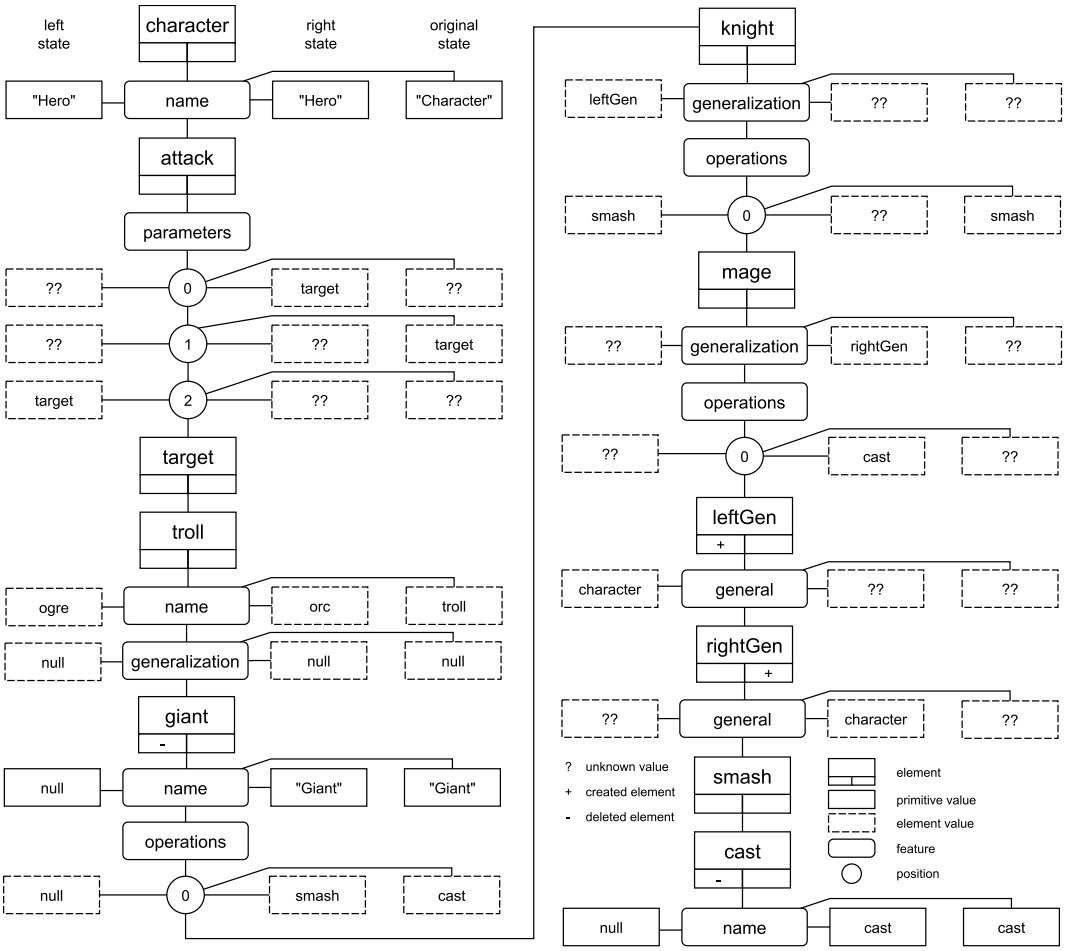


Figure 7.4: An element tree constructed from information in CBPs in Listings 7.2 and 7.3 (all left and right change events).

moves cast from giant's operations to mage's operations. From this move event, we can see that 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 see that Alice created a Generalization with ID rightGen. Thus, in elementTree, an element with ID rightGen is created. Since it has just been created in the active session, the element is marked with a '+' sign in elementTree on the right side. At line 37, we can also see 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 changes character's name from "Character" to "Hero". Since character and its feature name already exist in elementTree, we set name's value to "Hero" only on the right side. The original value was already 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 see that rightGen is no longer in troll's generalization but exists in mage's generalization. Since it is the first time 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 7.4 exhibits the state of the elementTree after both sides' change events have been processed.

Construction Procedure

The construction of elementTree follows the steps shown in Figure 7.5. 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 about 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, which was initialised before using the information I_{OL} (step 4), implying that $I_{OL} \cup I_{RR} \rightarrow S_R$. Also, information about the original model from the right change events I_{OR} is used to update the original state (step 5). Thus, we have constructed a partial state of the original model 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 7.5 in a generic fashion. It iterates through all of a model's change events and uses the information contained in

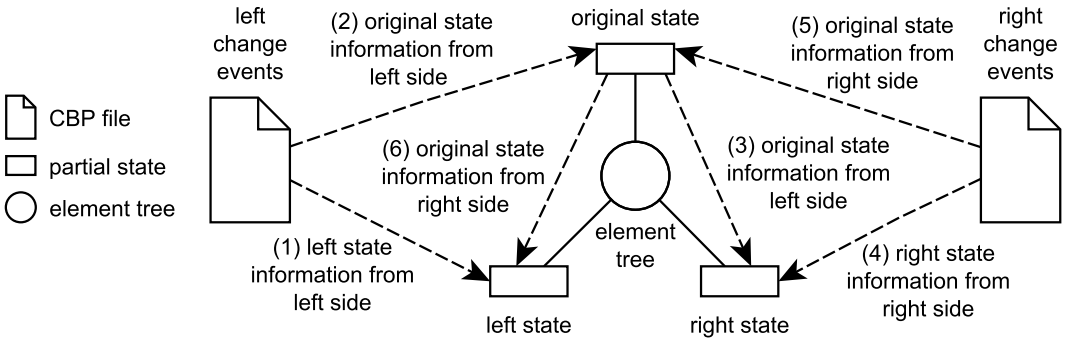


Figure 7.5: Steps in Element Tree construction.

them to construct the relevant partial state. The choice to begin with left or right change events 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. Then it 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 7.2). This targetElement—an instance of class Element in Figure 7.2—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 7.2. 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 7.2—represents the target element's feature (e.g., name and operations in Listing 7.3) 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 a type of Element (e.g., element leftGen line 36 in Listing 7.2) or primitive (e.g., the string "Hero" at line 34 in Listing 7.2). 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 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 modify only a target element not an element feature.

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
```

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 determine that the index of cast is 0 (line 35 in Listing 7.3) because the change event type is add. In a remove change event, we can get only the previousIndex of cast, which is 1 (line 35 in Listing 7.3), because the element does not exist anymore in the left model. We can obtain both of them only in a move change event as an element is moved from a previous index to a new one (line 31 in Listing 7.3). For a single-valued feature, the index and previousIndex are always 0, because the feature can contain only 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 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 does the same thing to the opposite side—if the current side is left then it is right.

7.4.3 Diff Computation

Using the elementTree presented in Figure 7.4, we can determine the difference between the left and right models without having to compare all their elements and features. After the elementTree has been constructed, we iterate through elements and features of the elementTree and use the flags, containers, containing features, and indexes on both sides of each element and value to identify differences between the left and right models. We follow the steps in Algorithm 2. The algorithm visits each element and every index of each feature (lines 3–5). At every index, it retrieves the leftValue and rightValue (lines 5–7), passing these, together with the element, feature, and index to a function identifyDiffUsingRules (line 8). The function uses a set of pre-defined rules to identify the differences diffs based on the states of flags of an element, flags and attributes of the element's feature, values of the feature,

and indexes of the values. The obtained diffs are then added to the overall list of differences diffList which is output (line 8–9, 13).

Algorithm 2: Algorithm to determine differences.

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

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

The first rule (Rule 1) in Algorithm 3 is to identify changes in single-valued attributes. A feature must be of type attribute, both side values must be different, and the element should have not been created or deleted in both models. The second rule (Rule 2) identifies whether an element is in a different location in the two models. The element must not have been deleted, and it must exist from the previous version—the original model. Also, the containers, containing features, or indexes of the element must be different on the two sides. The third rule (Rule 3) identifies the deletion of an element. If an element in the left model is not created but exists in the model, it means that the element has existed since the previous version—the original model.

Algorithm 3: Some rules to determine differences (part 1).

```
input  : an Element element, a Feature feature, a variable leftValue, a variable
         rightValue, an Integer index
output : a List of Diff diffs
1  diffs ← createDiffList();
   // ...
   // Rule 1:  a rule to determine a change of a single-valued
      attribute
2  if getType(feature) is Attribute and isSingleValued(feature) and leftValue <>
   rightValue and not leftIsCreated(element) and not leftIsDeleted(element) and not
   rightIsCreated(element) and not rightIsDeleted(element) then
3  |  diff ← createNewDiff(element, element, feature, feature, index, index, leftValue,
   |    rightValue, DifferenceType.CHANGE);
4  |  addDiffToDiffList(diff, diffs);
5  end
   // Rule 2:  A rule to determine movement of an element for right
      value (the left value has its own rule)
6  if getType(feature) is Containment and not isNull(rightValue) and not
   leftIsCreated(rightValue) and not leftIsDeleted(rightValue) and not
   rightIsCreated(rightValue) and not rightIsDeleted(rightValue) and
   (getLeftContainer(rightValue) <> getRightContainer(rightValue) or
   getLeftFeature(rightValue) <> getRightFeature(rightValue) or
   getLeftIndex(rightValue) <> getRightIndex(rightValue)) then
7  |  diff ← createNewDiff(getLeftContainer(rightValue), getRightContainer(rightValue),
   |    getLeftFeature(rightValue), getRightFeature(rightValue), getLeftIndex(rightValue),
   |    getRightIndex(rightValue), rightValue, rightValue, DifferenceType.MOVE);
8  |  addDiffToDiffList(diff, diffs);
9  end
   // Rule 3:  The first of two rules to determine the deletion of an
      element
10 if getType(feature) is Containment and not leftIsCreated(rightValue) and
   leftIsDeleted(rightValue) and not rightIsCreated(rightValue) and not
   rightIsDeleted(rightValue) then
11 |  createNewDiff(getLeftContainer(rightValue), getRightContainer(rightValue),
   |    getLeftFeature(rightValue), getRightFeature(rightValue), null,
   |    getRightIndex(rightValue), null, rightValue, DifferenceType.DELETE);
12 |  addDiffToDiffList(diff, diffs);
13 end
   // ...
   // continue to part 2
```

This also means that the element also exists in the right model, unless it has been deleted. Thus, to make the right model equal to the left model, the element must be deleted in the right model as well.

Algorithm 4: Some rules to determine differences (part 2).

```
   // continuation of part 1
   // ...
   // Rule 4:  The second of two rules to determine deletion of an
      element
1  if getType(feature) is Containment and not leftIsCreated(rightValue) and not
     leftIsDeleted(rightValue) and rightIsCreated(rightValue) and
     rightIsDeleted(rightValue) then
2      createNewDiff(getLeftContainer(rightValue), getRightContainer(rightValue),
         getLeftFeature(rightValue), getRightFeature(rightValue), null,
         getRightIndex(rightValue), null, rightValue, DifferenceType.DELETE);
3      addDiffToDiffList(diff, diffs);
4  end
   // Rule 5:  one of rules to determine addition of an element
5  if getType(feature) is Containment and leftIsCreated(leftValue) and not
     leftIsDeleted(leftValue) and not rightIsCreated(leftValue) and not
     rightIsDeleted(leftValue) then
6      diff ← createNewDiff(getLeftContainer(leftValue), getRightContainer(leftValue),
         getLeftFeature(leftValue), getRightFeature(leftValue), getLeftIndex(leftValue),
         null, rightValue, null, DifferenceType.ADD);
7      addDiffToDiffList(diff, diffs);
8  end
   // ...
9  return diffs
```

The fourth rule (Rule 4) in Algorithm 4 also identifies the deletion of an element. The element never existed in the left model, but it has been created in the right model. Thus, to make the right model equal to the left model, the element must be deleted from the right model. The fifth rule (Rule 5) identifies the need to add an element. If an element is created in the left model and has not been deleted, it means that the element should be added to the right model to make the two models equal.

In Figure 7.4, when the iteration of elementTree, from element character down to feature name of element cast reaches index 0 in feature parameters of element attack, we can see that rightValue has the value element target and the value of leftValue is

unknown. The rightValue is not null and value target exists on both sides—all its *Created and *Deleted flags are false, and it also different indexes (2 in the left state and 0 in the right state). This meets the condition of the second rule. Thus, we can conclude that, to make the index of element target in the right model equal its index in the left model, element target should be moved from index 0 to 2. Thus, the type of this difference is MOVE. We denote this difference as dc_1. The same rule is applied to element smash when the iteration reach index 0 in knight's generalization. Applying the rule to the element produces difference dc_3.

When the iteration is at feature name of element troll, we determine that the type of the feature is a single-valued attribute and the sides of the feature are different in value. This means that the condition of the first rule is met. Thus, we can conclude that, to make the left value of the feature equal to the right value, we must override the value "Orc" with "Ogre". The type of this difference is CHANGE. We denote this difference as dc_2.

At giant, the element used to exist but it has been deleted from the left model (flags leftIsCreated = false, leftIsDeleted = true); it still exists in the right state (flags rightIsCreated = false, rightIsDeleted = false). This condition satisfies the third rule. Therefore, element giant should be deleted from the right model. The type of this difference is DELETE. We denote this difference as dc_4. The same rule is applied to element cast when the iteration reaches the element. Applying the rule to the element produces difference dc_6.

We can get only one value when the iteration is at index 0 in the element knight's feature generalization; the leftValue is element leftGen, but the rightValue is unidentified. Thus, we process only the leftValue. Element leftGen is created only in the left model (flags leftIsCreated = true, leftIsDeleted = false, rightIsCreated = false, rightIsDeleted = false). This meets the condition of the fifth rule. Thus, to make element leftGen exist in the right state, we must add it into element knight's feature generalization at index 0. Therefore, the type of this difference is ADD. We denote this difference as dc_7.

When the iteration is at index 0 in the element mage's feature generalization, we can get only one value; the leftValue is unidentified and the rightValue is element

rightGen. Therefore, we process only the rightValue. Element rightGen is created only in the right model (flags leftIsCreated = false, leftIsDeleted = false, rightIsCreated = true, rightIsDeleted = false). This meets the condition of the fourth rule. Thus, to make element rightGen cease to exist in the left state, we must delete it from index 0 in element mage's feature generalization. Therefore, the type of this difference is DELETE. We denote this difference as dc_5.

Similar to the state-based approach in Section 7.3, we express identified differences as $dc_n = [LeftContainer_n, RightContainer_n, LeftFeature_n, RightFeature_n, LeftIndex_n, RightIndex_n, LeftValue_n, RightValue_n, Kind_n]$. Thus:

dc_1 = [attack, attack, parameters, parameters, 2, 0, target, target, MOVE]
dc_2 = [troll, troll, name, name, 0, 0, "Ogre", "Orc", CHANGE]
dc_3 = [knight, giant, operations, operations, 0, 0, smash, smash, MOVE]
dc_4 = [resource, resource, null, null, null, 2, null, giant, DELETE]
dc_5 = [mage, mage, generalization, generalization, null, 0, null, rightGen, DELETE]
dc_6 = [mage, mage, operations, operations, null, 0, null, cast, DELETE]
dc_7 = [knight, knight, generalization, generalization, 0, null, leftGen, null, ADD]

This change-based approach might produce differences that are distinct from differences identified using state-based approaches. This can be seen by comparing ds_1 and dc_1 ($ds_1 \neq dc_1$, [attack, attack, parameters, parameters, 0, 1, gem, gem, MOVE] $\neq$ [attack, attack, parameters, parameters, 2, 0, target, target, MOVE]). The state-based approach identifies element gem as the element that should be moved to index 0 to resolve the differences in attack's parameters (ds_4), while in the change-based approach, the difference is attributed to element target (dc_4). However, in both approaches, if we resolve their differences by performing all-left-to-right merging—making the right model equal to the left model, the two approaches produce models that are equivalent. In this way, we can check the correctness of the identified differences produced by the change-based approach.

7.5 Evaluation

This section presents the method used to evaluate the proposed change-based model differencing approach as well as the evaluation results.

7.5.1 Method

To assess the performance benefits of the change-based approach in terms of model differencing, we have evaluated it against a mature and widely used state-based comparison tool (EMF Compare [30,70]). Since there are no large, manually developed models persisted in our change-based format yet, the dataset for our experiments was constructed from a large model reverse-engineered from the Eclipse Epsilon project [64,65]. This model conforms to the Java meta-model [76], and it consists of more than 1.6 million elements with a size of 224 MB when persisted in XMI.

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

We conducted five experiments. In the first experiment, the ratio of occurrence between add, remove, move, and set changes was set to 1:1:20:40. This reflects an assumption that in a mature model, modification—move and set events—occurs more frequently than addition and deletion. So the change of total elements does not affect our measurement, the number of total elements should be kept constant. For example, it is difficult to determine if 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 is to exclude add and remove operations. However, excluding both operations made measurement less representative. Thus, we included both operations, but we made their probabilities equal so that the number of total elements remains largely unchanged. In the rest of the experiments, we performed only homogeneous operations—isolated from other types—per experiment (e.g., add-only, move-only

operations). In the end, we obtained five results: mixed, add-only, remove-only, move-only, and set-only measurements. We did this to assess whether operations of different types have different impacts on model differencing.

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

7.5.2 Results and Discussion

This section reports on the results for comparison time and memory footprint for the mixed and homogeneous operation experiments.

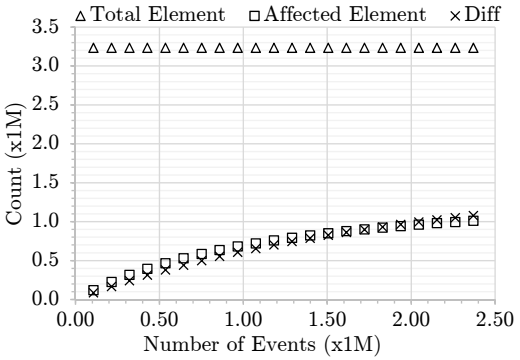
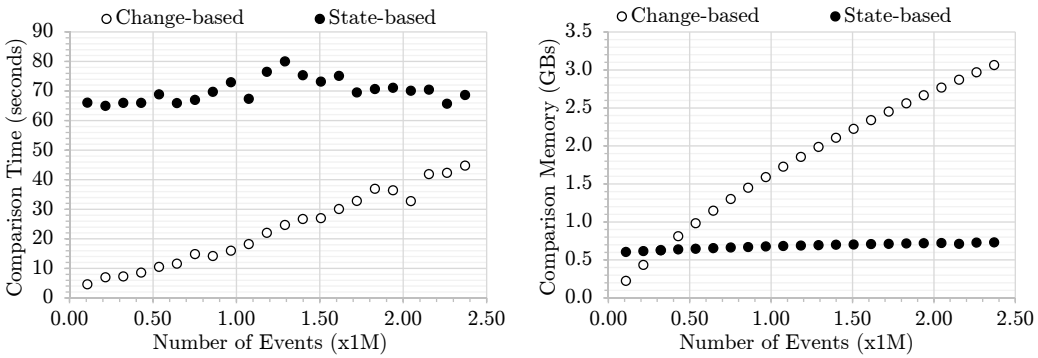


Figure 7.6: total elements, affected elements, and diffs

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 are shown in Figure 7.6. The growth is logarithmic since the probability that the random operations modify the same elements also increases. Thus, some change events might not add new affected elements and differences. In other words, more events are required to increase the number of affected elements or differences. In Figure 7.6, the total number of elements remains largely unchanged because the probabilities of addition and deletion were made equal, as noted in Section 7.5. The figure gives us an insight

about the characteristics of the modification caused by the random operations in the mixed operation measurement; it helps to explain the implications of the changes on execution time and memory footprints of model differencing.



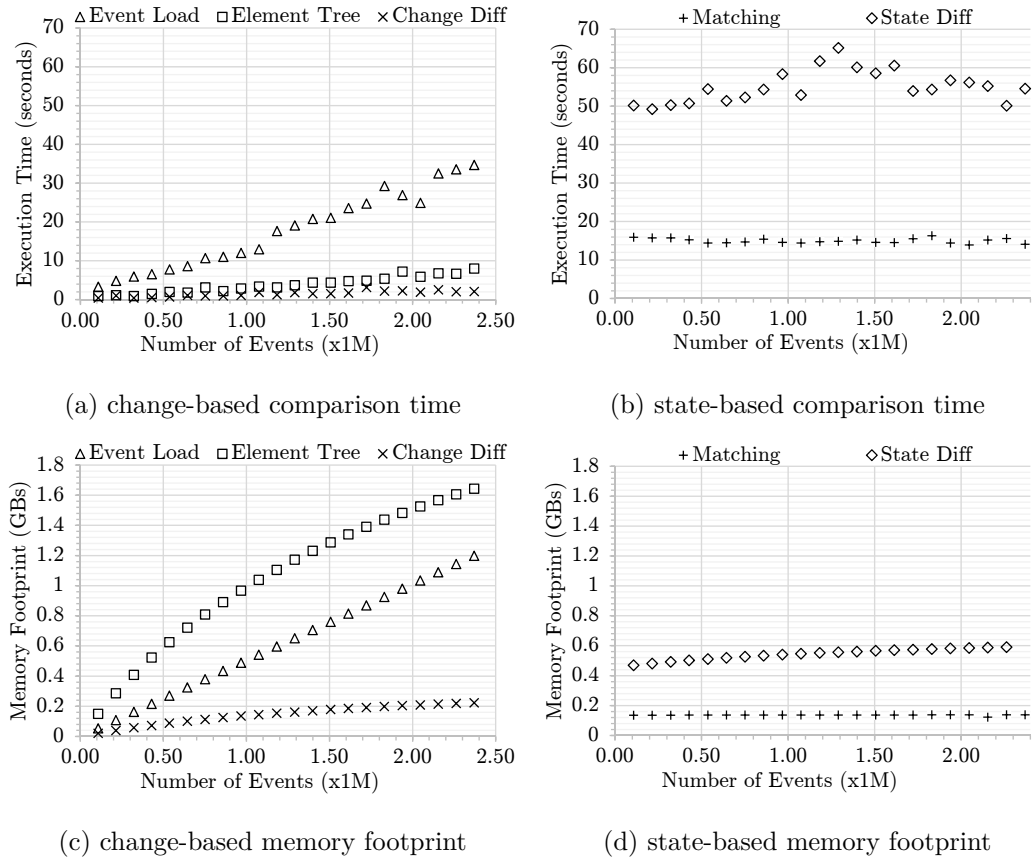
(a) execution time (b) memory footprint

Figure 7.7: Change-based vs. state-based model differencing as differences increase.

After applying some random changes on both models, the modification produces 100,000 change events at the first measurement point. Using this amount of events, our change-based comparison takes only 5 seconds to identify around 90,000 differences, in contrast to state-based comparison, which takes 66 seconds (see the first measurement points in Figures 7.6 and 7.7a). If the modification continues, more change events are generated. This growing number of change events must be loaded into memory and thus slows down the change-based comparison. Nevertheless, change-based comparison is still faster than state-based comparison. Even when the number of change events reaches 2.37 million—more than 1 million differences change-based comparison outperforms state-based comparison in execution time (Figure 7.7a). Figure 7.8a presents the comparison time in detail. It shows that the event loading time is the dominant contributor to the slowdown compared to the element tree's construction time and diffing time.

For the state-based comparison in Figure 7.8b, the comparison time experiences only a slight increase as the number of identified differences also grows. This slight increase comes mainly from the diffing time, while the matching time tends to be constant because of the very small increase of total elements (Figures 7.6).

Nevertheless, a change-based comparison generally consumes more memory than a state-based comparison (see Figure 7.7b). It consumes less memory than its state-



(a) change-based comparison time

(b) state-based comparison time

(c) change-based memory footprint

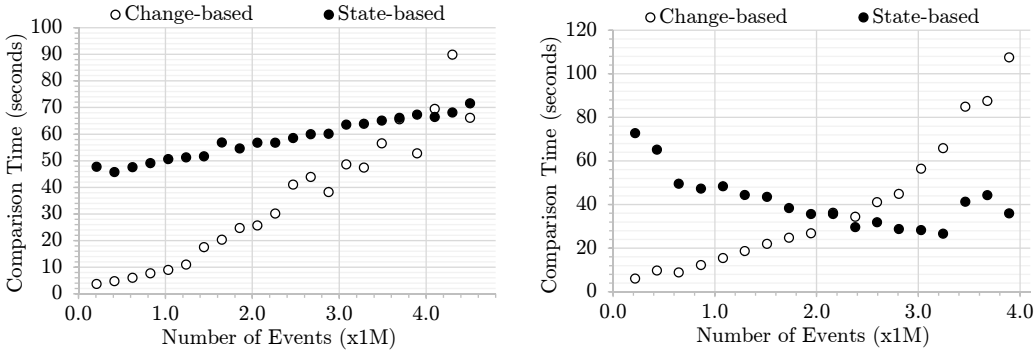
(d) state-based memory footprint

Figure 7.8: Breakdown view of comparison time and memory footprint in Figure 7.7.

based counterpart only when the number of events is fewer than 0.3 million. (At that moment there are fewer than 0.25 million identified differences.) Figure 7.8c separates the memory footprint of the change-based comparison into three factors: the loaded change events, element tree, and diffs. As modification continues, more events are generated. These events must be loaded into memory since they contain the information needed to construct an element tree. The amount of space to keep these change events in memory grows linearly with their number.

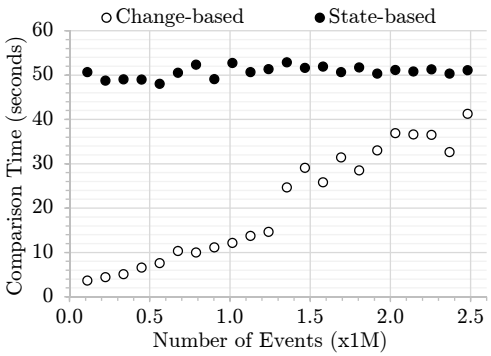
In contrast, the memory used for the element tree grows logarithmically. As the number of events increases, the probability that events modify already affected elements also increases. Thus, no additional memory allocation is required for the element tree. Moreover, the element tree occupies most of the memory footprint since it mirrors the partial states—elements, features, and values—of the models that are affected by the changes. In our technical implementation, a feature can have many instances—one instance for each element. (As a comparison, in the EMF

implementation, there is only one instance for a feature. The feature is used as a key so that different elements can have the same feature that maps to different values simultaneously). This contributes to the large memory footprint used by the element tree. The identified change-based diffs, the third factor, are the smallest factor that contributes to the memory footprint of the change-based comparison.

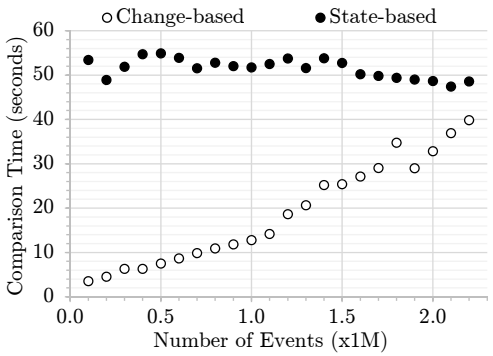


(a) add-only

(b) delete-only



(c) move-only



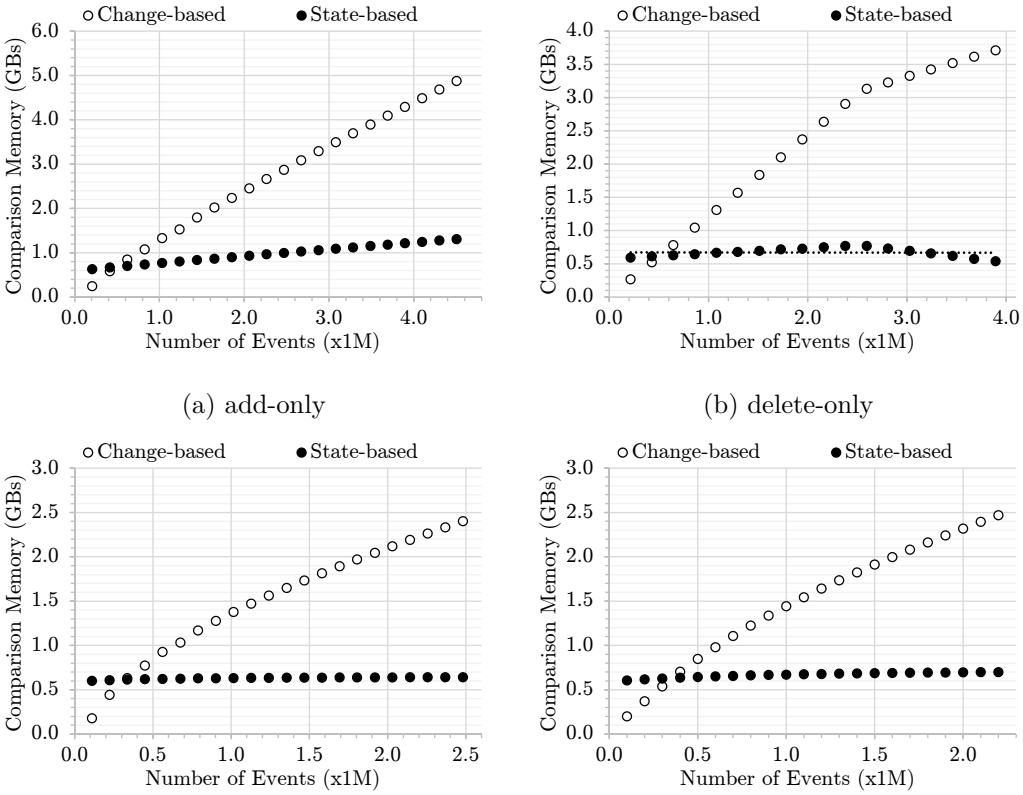
(d) change-only

Figure 7.9: Comparison time for homogeneous operations.

For the state-based comparison in Figure 7.8d, the memory footprint grows only slightly with the increase of differences. A large part of the memory footprint is used to represent the identified differences, while the memory used for matches tends to be constant, because the changes of the total elements are very few—fewer new elements means less memory must be allocated for new matches (Figures 7.6).

Homogeneous Operations

Figures 7.9 and 7.10 show the comparison times and memory footprints of models modified using homogeneous operations—add, remove, move, or set only. In all



(a) add-only

(b) delete-only

(c) move-only

(d) change-only

Figure 7.10: Memory footprint for homogeneous operations.

these figures, change-based comparison outperforms its state-based counterpart, particularly when the number of change events is small relative to the size of the model. As the number of modifications grows, change-based comparison becomes slower than state-based comparison. In our experiments, this happens when the number of events is greater than 4 million (Figure 7.9a). Change-based comparison also becomes slower when the size of models shrinks (because of a large number of delete events) as depicted in Figure 7.10b. This is because change-based comparison still needs to load these change events and construct its element tree. In contrast, deletion means less work for state-based comparison. In terms of memory footprint, change-based comparison performs better than state-based comparison only when the number of change events is fewer than 0.3 million, as depicted in Figure 7.10.

7.6 Conclusions

This chapter proposed an approach to identify differences between two versions of a model persisted in change-based format. It works by loading into memory the changes made to both versions since the last shared version, constructing partial states of the versions based on the information in the latest changes, and using specific rules to identify differences between the versions' elements and features.

The evaluation indicates that the change-based comparison approach works best for large models that have been modified a moderate number of times. Models that have been modified excessively and experience a significant reduction in size could impair the performance of change-based model differencing, since a high number of change records must be read and loaded into memory.

This chapter has addressed the second research question of this study, **In a changed-based format, how can the differences between models be identified, and how does change-based model differencing perform, in terms of speed and memory footprint, compared to state-based model differencing?** (RQ2). Change-based persistence can identify differences between two versions of a model. The change-based representation of the two versions contains all the information needed to identify elements that have been modified since their last shared version. In this way, we can localise the model differencing to the elements that were modified recently. In other words, it is not necessary to inspect, match, and diff all the elements. We can reconstruct the partial states of the two versions and then compare their elements and features using specific rules to identify their differences.

The change-based model differencing proposed in this research comprises three phases: event loading, element tree construction, and diff computation. In the event loading phase, the implementation loads the change events recorded in two change-based model persistence files into memory starting from the line their change events are different. The information that the loaded change events contain is used to construct an element tree. An element tree contains only the affected elements and features of the versions being compared, including the shared original version. This is possible because change events are designed to contain adequate information to construct the element tree. A diff computation is then executed to identify the differences using a

set of pre-defined rules (i.e., if an element is created in one version it means that the element does not exist in the other version or in the original version).

The evaluation suggests that the proposed change-based model differencing executes faster than traditional, state-based model differencing. However, change-based model differencing needs to load change events from a change-based persistence into main memory. Thus, it may require more memory than is needed for state-based model differencing. In our evaluation, this occurs when the number of change events exceeds 400,000. However, it is likely that diff and merge operations are performed on lower numbers of changes (smaller deltas) than were tested in this evaluation.

Chapter 8

Efficient Conflict Detection of Change-based Models

In Chapter 7, it was demonstrated that change-based model persistence can be used to speed up model differencing. This chapter explores whether change-based model persistence can be leveraged to improve conflict detection in model versioning. Results show that the proposed approach can reduce conflict detection time (up to 90% in some experiments) compared to existing state-based and change-based conflict detection approaches.

8.1 Introduction

State-based and change-based model conflict detection are discussed briefly in Sections 8.2 and 8.3. The state-based approach, represented by EMF Compare [30], does have drawbacks. First, it cannot detect conflicts as accurately as a change-based approach can. This is because their changes are derived; they are not working with real historical changes. Second, EMF Compare uses a three-way model comparison [30]. Therefore, its conflict detection should perform somewhat more slowly than the change-based approach, since it has to perform state-based model differencing twice. It must derive change events between left and original versions and between right and original versions.

Change-based model conflict detection [77], represented by EMF Store [38], also has

drawbacks. EMF Store works only on change events, and it detects conflicts based on pre-defined rules; it does not consider the eventual states of two versions that are being compared. Thus, two change events that modify a same feature are considered in conflict even though both change events produce the same eventual states. In terms of performance, as has been presented in Chapter 7, the change-based approach is faster than its state-based counterparts in model differencing. Thus, it is expected that it can also perform better than the state-based approach in detecting conflicts.

This chapter introduces a proposed change-based approach to detect conflicts between two versions of a model, based not only on recent change events of the two versions but also by considering the eventual states of the elements affected by the change events. Thus, the performance and accuracy of model conflict detection can be improved compared to existing state-based and change-based approaches represented by EMF Compare and EMF Store, respectively.

The rest of this chapter is structured as follows. Sections 8.2 and 8.3 provide an overview of conflict detection by EMF Compare and EMF Store, respectively. Sections 8.4 and 8.5 discuss our proposed approach to detect conflicts and review its accuracy compared to EMF Compare and EMF Store. Section 8.7 reports the results of experiments used to evaluate the proposed approach. Section 8.7.3 concludes this chapter.

8.2 State-based Conflict Detection (EMF Compare)

In this study, we select EMF Compare [30] as an example to explain conflict detection in state-based model persistence. We also use it as a benchmark in the comparative evaluation of this paper. It is selected because of its maturity and ongoing development activity—4,682 commits and 103 releases on GitHub [78]. Another implementation of state-based conflict detection is EMF DiffMerge [50]. However, its comparison approach is similar to EMF Compare [50], and it is less mature than EMF Compare—only 442 commits and 20 releases on GitHub [79].

In state-based model comparison, a conflict occurs when the states of an element or a feature are different in the versions of a model that are being compared. In other words, the change events that cause the differences are in conflict, since they

produce two different states. State-based persistence does not record change events that cause the differences. Thus, the change events must be identified through model differencing [15, 30].

Let's say that we have three versions of model M, 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 identified change events, left change events C_L and right change events C_R. These sets are obtained by differencing m_l to m_o and m_r to m_o using an LCS (Longest Common Subsequence) algorithm [24, 30], where $C_L = \{c_{l1}, c_{l2}, ..., c_{olm}\}$, $C_R = \{c_{r1}, c_{r2}, ..., c_{rn}\}$, m is the number of left change events in C_L or $m = |C_L|$, and n is the number of change events in C_R or $n = |C_R|$. 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 used to detect conflicts using Equations (8.1), (8.3), and (8.2).

If state-based model differencing is used to derive left change events C_L from the left and original versions (Bob's and Jane's versions) in Figure 4.2, the following change events are obtained.

Listing 8.1: The derived, minimal change events to produce the left version (Bob's version) in Figure 4.2b from the original version (Jane's version).

```
1  move target in attack.parameters from 1 to 2
2  set character.name from "Character" to "Hero"
3  set troll.name from "Troll" to "Ogre"
4  create leftGen type Generalization composite l1
5  set leftGen.general from null to character composite l1
6  set knight.generalization from null to leftGen composite l1
7  unset cast.name from "cast" to null composite l2
8  remove cast from giant.operations at 0 composite l2
9  delete cast composite l2
10 unset giant.name from "Giant" to null composite l3
11 remove giant from resource at 2 composite l3
12 delete giant composite l3
```

And the 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 4.2.

Listing 8.2: The derived, minimal change events to produce the right version (Alice's version) in Figure 4.2c from the original version (Jane's version).

```
1  move gem in attack.parameters from 0 to 1
2  set character.name from "Character" to "Hero"
3  set troll.name from "Troll" to "Orc"
4  remove smash from knight.operations at 0 composite r1
5  add smash to giant.operations at 0 composite r1
6  create rightGen type Generalization composite r2
7  set rightGen.general to character composite r2
8  set mage.generalization to rightGen composite r2
9  remove cast from giant.operations at composite r3
10 add cast to mage.operations at 0 composite r3
```

Both Listings 8.1 and 8.2 are derived change events. They are the minimal sequences of change events that can produce m_l and m_r from m_o respectively, but not necessarily the real changes made by Bob and Alice. For example, Bob and Alice might have created and then deleted a new class in the process, or they might have modified a feature but later decided to set it back to its initial value.

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 (8.1). EMF Compare [30] classifies this conflict as a REAL conflict. For example, Bob changed the name of troll to "Ogre" (Listing 8.1) while Alice modified it to "Orc" (Listing 8.2).

$$e_o + c_l \not\equiv e_o + c_r \Rightarrow c_l \,!\, c_r \tag{8.1}$$

Non-applicability. A REAL conflict also occurs when applying 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 (8.2). For instance, Alice moved operation smash from class Knight to class Giant (Listing 8.2), but this class was deleted by Bob (Listing 8.1). Deleting class Giant makes the move inapplicable.

$$(e_o + c_r \not\equiv e_o) \wedge (e_o + c_l + c_r \equiv e_o + c_l) \Rightarrow c_l \,!\, c_r \tag{8.2}$$

Pseudo Conflict. A conflict is classified as PSEUDO if the eventual states produced are equivalent. PSEUDO means the conflict can be automatically resolved by choosing any of the conflicting changes, since any of the changes produces the same eventual state (8.3) [30]. 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 8.1 and 8.2).

$$e_o + c_l \equiv e_o + c_r \Rightarrow c_l \; !_p \; c_r \tag{8.3}$$

Table 8.1: Conflicting change events identified by EMF Compare based on the case in Figure 4.2.

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, non-applicability
EC4	delete giant	remove smash from knight.operations at 0 add smash to giant.operations at 0	real, non-applicability

Using Equations (8.1), (8.2), and (8.3) and information in Listings 8.1 and 8.2, four conflicts can be identified. They are presented in Table 8.1 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. 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 [30] does not detect that Alice and Bob modified the same element—parameter target—as indicated by line 29 in List. 7.3 and line 35 in List. 7.2. 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 (8.1), the two change events are not in conflict since these change events modify two different elements, target and gem. The result is different if a change-based approach is employed to detect

conflicts using the change event records in Listings 7.2 and 7.3. This is explained in Section 8.3.

8.3 Change-based Conflict Detection (EMF Store)

EMF Store [11] is an open-source tool 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 [38]. EMF Store uses the following rules to identify conflicts between change events [77].

Non-commutability. In EMF Store, change events c_l and c_r are in conflict if applying them in different order to the same element e_o produces two different eventual states [77]. For example, Alice changed the name of class Troll to "Orc" (Listing 7.3), while Bob renamed it to "Ogre" (Listing 7.2). Applying Alice's change first to Bob's change makes the class's name "Ogre", but applying Bob's change first results in "Orc".

$$e_o + c_l + c_r \not\equiv e_o + c_l + c_r \Rightarrow c_l \,!\, c_r \tag{8.4}$$

However, after examining the implementation [80], even though two different change events produce equivalent eventual states, both change events are still treated as conflict by EMF Store (8.5). For example, both Bob and Alice changed the name of element character from "Character" to "Hero" (Listing 7.2 line 34 and Listing 7.3 line 39). The reason is that, 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 the previous value "Character" is inapplicable since it makes the sequence of the change events inconsistent. Bob's set event produces the eventual value "Hero", which is not the previous value changed by Alice's set event, which is "Character". The same inconsistency occurs even we apply these set events in a different order.

$$e_o + c_l + c_r \equiv e_o + c_l + c_r \Rightarrow c_l \,!\, c_r \tag{8.5}$$

Moreover, a conflict occurs even when two different sets of change events, C_L and C_R, produce eventual states that are equal to their initial states (8.6). For example, if both Bob and Alice alter character's name from "Character" to "Hero" and then

modify it back to “Character”, both sets of change events are also treated in conflict.

$$e_o + C_L + C_R \equiv e_o + C_R + C_L \equiv e_o \Rightarrow C_L \,!\, C_R \tag{8.6}$$

Co-modification. This leads to a new definition that a conflict occurs when two different change events modify the same element or feature regardless of the eventual state that they produce.

$$(e_o + c_l \equiv e_o + c_r) \vee (e_o + c_l \not\equiv e_o + c_r) \Rightarrow c_l \,!\, c_r \tag{8.7}$$

Non-applicability. This non-applicability rule is the same as the non-applicability rule in state-based conflict detection. Essentially, a conflict occurs when applying 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 7.3), but this class was deleted by Bob (Listing 7.2). Deleting class Giant makes Alice's move inapplicable.

$$(e_o + c_r \not\equiv e_o) \wedge (e_o + c_l + c_r \equiv e_o + c_l) \Rightarrow c_l \,!\, _r \tag{8.8}$$

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, deleting class Giant is part of composite event l2 (Listing 7.2) and adding operation smash to class Giant is part of composite event r1 (Listing 7.3). Since they are in conflict according to (8.8), all other change events in their composite events, l2 and r1, also are in conflict.

$$c_l \,!\, c_r \wedge c_r \in cc_r \Rightarrow c_l \,!\, \forall c_{rn} | c_{rn} \in cc_r \tag{8.9}$$

In change-based conflict detection, all change events applied to a model are readily available. Thus, there is no need to derive change events through 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. Therefore, change-based conflict detection can detect conflicts that cannot be detected by state-based conflict detection. For example, in Listing 7.3 line 31, parameter target has been moved from index 1 to 0, while in Listing 7.2 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 as being in conflict using (8.7). The same parameter target is modified by two different change events.

Table 8.2: Conflicting change events identified by EMF Store in Listings 7.3 and 7.2.

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-modification, composite
ES2	set character.name from "Character" to"Hero"	set character.name from "Character" to "Hero"	co-modification
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-modification

The drawback of EMF Store is that it considers two change events to be in conflict if they modify the same element but create the same end state of the element [56]. 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 does not classify conflicts as REAL or PSEUDO, in EMF Compare does, to automate conflict resolution.

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 8.2, ES1). Using the co-modification (8.7) rule, we can determine that the setting and unsetting of troll's generalization to leftGen and null (Listing 7.2 lines 33, 35) are in conflict with the setting and unsetting of troll's generalization to rightGen and null (Listing 7.3 lines 38, 40). Moreover, using the composite (8.9) rule, we can also determine that the setting of knight's generalization to leftGen (Listing 7.2 line 36) and the setting of mage's generalization to rightGen (Listing 7.3 line 41) are also part of conflict ES1, since both events are in the same composite move events, l1 and r3, with the unsetting of troll's generalization to null (Listing 7.2 line 35, Listing 7.3 line 38).

In state-based conflict detection, case ES1 is not a conflict since the values of class troll's feature generalization in Jane's, Bob's, and Alice's versions are identical—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-applicable, 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 can be detected with the co-modification (8.7) rule. The states of troll's name have been simultaneously modified to "Ogre" or "Orc".

Table 8.3: 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 - in large models with moderate changes, it should perform faster than the state-based approach—no need to derive changes since they are already available
Drawbacks	- less accurate in detecting conflicts since changes are derived—not real changes - in large models, its performance should be slower than the change-based approach since it performs a three-way comparison, which requires two-times model differencing to derive changes - in small models, it should perform faster than change-based approach	- treats all conflicts as REAL conflicts which demand user intervention for resolution - can be oversensitive in detecting conflicts since eventual states are not considered - in small models with excessive changes, it should perform more slowly than the state-based approach because it must process many change records

8.3.1 Summary

The summary of the advantages and drawbacks of EMF Compare and EMF Store in detecting conflicts are presented in Table 8.3. The state-based approach, represented by EMF Compare [30], does have drawbacks. First, it cannot detect conflicts as accurately as can change-based approaches because it uses derived changes—not real

historical changes. Second, EMF Compare uses a three-way model comparison [30] thus hypothetically its conflict detection should perform more slowly than the change-based approach, since it must perform state-based model differencing twice to derive change events: change events between the left and original versions, and change events between the right and original versions.

Change-based model conflict detection [77], represented by EMF Store [38], also has drawbacks. EMF Store works only on change events, and it detects conflicts based on pre-defined rules; it does not consider the eventual states of the versions that are being compared. Thus, two change events that modify the same feature are considered to be in conflict even though both change events produce the same eventual state. This can make EMF Store oversensitive in conflict detection.

8.4 EMF CBP Conflict Detection

The model conflict detection procedure proposed in this study performs like the phases of change-based model differencing discussed in Chapter 7.4 but with some modification. First, the conflict detection still performs the event loading and element tree construction phases, but the diff computation phase is replaced by a conflict computation phase. Second, during element tree construction, the conflict detection maps change events to the elements, features, and values that the change events modify. The change event mapping and conflict computation are discussed in the following Sections.

8.4.1 Change Event Mapping

Using the information in the change-based model representations in Listings 7.2 and 7.3, we can construct an element tree as depicted in Figure 7.4 using the construction method presented in Section 7.4.2. During that construction, change events in Listings 7.2 and 7.3 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 7.2. 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 multi-valued-features. With this mapping, we can trace all events that affects certain elements, features, and values. The mapping of the events in Listings 7.2 and 7.3 is in Table 8.4. The application of this mapping is presented in Section 8.4.3.

Table 8.4: Mapping the elements, features, and values in Figure 7.4 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

8.4.2 Theoretical Foundation

To improve the accuracy of the proposed conflict detection approach, we take two strategies from both change and state-based conflict detections. First, we exploit change events to address real historical changes—not derived ones—of models. Second, we also take into account the original and eventual states of the 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 we do not need to calculate them again in the conflict computation phase. Since all change events are also recorded for 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 to change its state to element e_l and $g = |C_L|$.

$$e_o + c_{l1} + c_{l2} + ... + c_{lg} \rightarrow e_l \tag{8.10}$$

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

$$e_o + c_{r1} + c_{r2} + ... + c_{rh} \rightarrow e_r \tag{8.11}$$

Non-conflict. Instead of calculating conflict between change events, we start by checking the equivalence 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 changes 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 the eventual state. We also identify no conflict if an element is modified only on one side—no change events are applied on the other side.

$$\begin{aligned} &(e_o \equiv e_l \wedge e_o \equiv e_r) \vee |C_L| = 0 \vee |C_R| = 0 \Rightarrow \\ &\neg(\forall c_l \,!\, \forall c_r) \mid c_l \in C_L, c_r \in C_R \end{aligned} \tag{8.12}$$

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 there is at least one 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.

$$\begin{aligned} &(e_o \not\equiv e_l \vee e_o \not\equiv e_r) \wedge (|C_L| > 0 \wedge |C_R| > 0) \Rightarrow \\ &\forall c_l \,!\, \forall c_r \mid c_l \in C_L, c_r \in C_R \end{aligned} \tag{8.13}$$

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.

$$(e_o \equiv e_l \vee e_o \equiv e_r \vee e_l \equiv e_r) \wedge (|C_L| > 0 \wedge |C_R| > 0) \\ \Rightarrow \forall c_l \; !_p \; \forall c_r \mid c_l \in C_L, c_r \in C_R \tag{8.14}$$

Figure 8.1 illustrates 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 8.1a 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 (8.12). In Figure 8.1b, the element is modified on the right side (version) only. Thus, using (8.12), 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 8.1c, even though an element has been modified on both sides, using (8.12), no conflict is detected, since both left and right states are equal to the original state after the modification. In the figure, both C_L and C_R produce eventual states that are equal to the original state, 'a'.

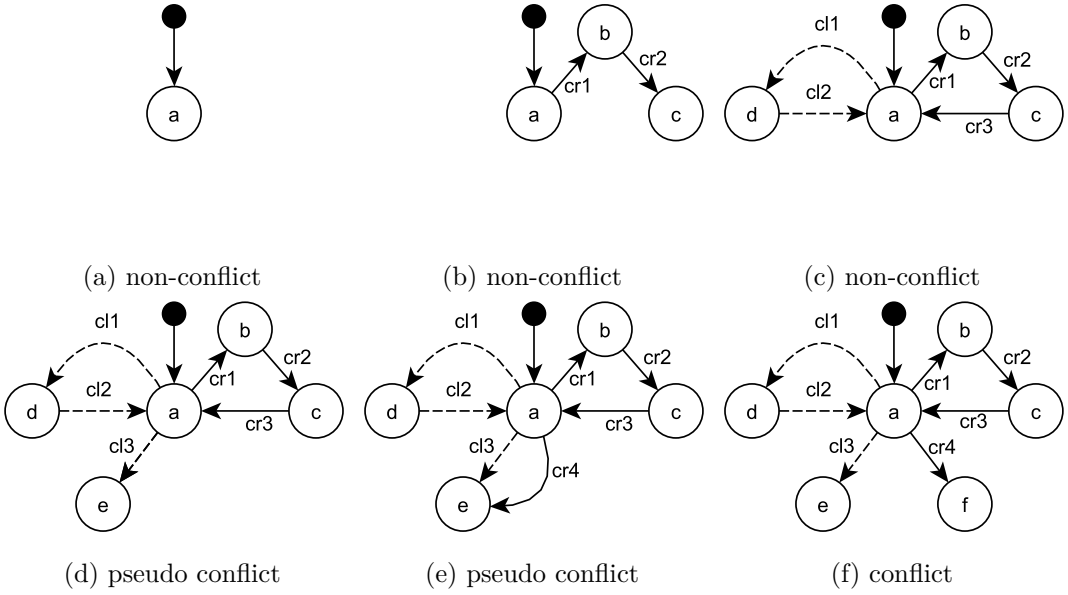


Figure 8.1: Conflicting and non-conflicting change events (dashed arrow = left change event, solid arrow = right change events, circle = state).

Using (8.14), the condition in Figure 8.1d 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 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 the change events in C_L are accepted to produce the eventual state, which is 'e'. Also using (8.14), the condition in 8.1e can 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 can be automatically resolved since selecting either one of the sets produces the same outcome. With (8.13), the scenario in Figure 8.1f can be detected as a REAL conflict, since change event sets, C_L and C_R, produce two different eventual states. The conflict cannot be automatically resolved, and it requires user intervention to choose which one is the desired eventual state, 'e' or 'f'. Then the appropriate change event set can be selected to produce the eventual state.

8.4.3 Conflict Computation

We perform the procedure in Algorithm 5 and use (8.13) and (8.14) inside it to identify conflicts between two CBPs. The algorithm iterates through all the elements, features, and values in the element tree (Figure 7.4), checks the equivalency of their original and eventual states, and records 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 an empty list conflictList to contain identified conflicts at line 2. The algorithm then iterates through all the elements, features, and values in the element tree.

Conflict with Deletion

At lines 4 to 11 in Algorithm 5, the algorithm checks if there is a conflict related to a deletion of an element. If an element is deleted on one or both sides, it means that all events related to that element on both sides should be in conflict. To get all the related events, the algorithm uses two functions, getAllRelatedLeftEvents(*element*) and getAllRelatedRightEvents(*element*). (The element acts as a map key to access the change events.) These functions return two sets of related events, leftEvents and

Table 8.5: Conflicting change events in Listings 7.2 and 7.3 identified by the proposed change-based conflict detection. The bold identifiers are the keys where conflicts were 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, non-applicability
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, non-applicability
CB5	set **character.name** from "Character" to "Hero"	set **character.name** from "Character" to "Hero"	pseudo

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 reaches element giant in Figure 7.4, the algorithm determines that the element has been deleted only on the left side. Using the map in Table 8.4, 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 the 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 the left-side change event at line 40 is collected. For key giant.operations.cast, it collects the left-side change event at line 39 and the right-side change event at line 34. For key giant.operations.smash, only the 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 and 35 for the right side. For key giant.name, only the left-side change event at line 38 is collected. The collected

Algorithm 5: 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
```

change events are merged into one set of change events for each side. So, the left events are all events that comprise the composite event that deletes the element. The right events are events that move operation 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 the events producing conflict CB3 in Table 8.5.

When the iteration reaches element cast—the operation of class giant, the same procedure is repeated. It collects left-side change events at lines 33, 38, 39, 40, 41, and 42, and right-side change events at lines 34, 35, and 38. The left-side change events related to element giant also are included since they are in one composite event that also affects element cast. These change events are collected into one conflict, CB4.

It should be noted that both conflicts CB3 and CB4 have shared change events. Thus, these conflicts have a dependency on each other. This means that if a user chooses to delete giant—chooses the left side as the solution—for conflict CB3, the left side change events also must be selected as the solution for conflict CB4 for consistency. To facilitate computing such dependencies, conflicts and change events are designed to have many-to-many relationships, as depicted in Figure 7.2. Thus, if a change event is associated with two or more conflicts, it means that they depend on each other.

It is important to notice that at line 13 in Figure 5 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 getAllRelatedLeftEvents(*element*) and getAllRelatedRightEvents(*element*) at lines 5 and 6.

Conflict between Cross-container Moves

Lines 15 to 25 in Algorithm 5 are dedicated to identifying 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 or both sides. If it has been moved, the algorithm then checks the number of events related to the element.

First, it obtains change events related to the element on both sides using functions getAllRelatedLeftEvents(*element*) and getAllRelatedLeftEvents(*element*). This yields 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 the element is moved to the same container on both sides or if the element is moved but then returns to its original container on one of its sides, then the conflict is set to PSEUDO.

Algorithm 6: Algorithm to handle single-valued features in conflict detection using an element tree—handleSingleValuedFeature (*element*, *feature*, *conflictList*) at line 27 in Algorithm 5.

```
   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
```

Single-valued Feature Conflict

Conflicts that involve single-valued features are handled by the procedure at line 28 in Algorithm 5, which is elaborated in Algorithm 6. 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 either leftValue or rightValue is not equal to originalValue, it continues to check the number of change events related to the feature by retrieving them using functions getAllRelatedEvents(*element*,

feature) and getAllRelatedRightEvents(*element*, *feature*). (Element and feature act as map keys to access the events.) This yields 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 whether leftValue and rightValue are equal, and it sets the conflict to PSEUDO if leftValue and rightValue are equal to each other or if one of them is equal to originalValue. Finally, the conflict is put into conflictList.

For example, when the iteration reaches 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 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 8.5. 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 the same value, "Hero".

Ordered Multi-valued Feature Conflict

Conflicts that involve multi-valued features are handled by the procedure at line 29 in Algorithm 5. The procedure is elaborated in Algorithm 7, where ordered multi-valued features are addressed at lines 3–15. The procedure relies on the function getUnequalLeftAndRightValues. This 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 7.4, 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 an index different from its original index, but it is already included in the returned set.

The algorithm then iterates through the values of the set. For each value, it retrieves all events related to the value of this feature. (Element, feature, and value act as map keys to access the events.) The algorithm uses function getAllRelated *Events(*element*,

Algorithm 7: Algorithm to handle multi-valued features in conflict detection using an element tree—handleMultiValuedFeature(*element*, *feature*, *conflictList*) at line 28 in Algorithm 5.

```
   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
```

feature, value), which yields 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. Finally, 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 is detected. This conflict is presented as conflict CB2 in Table 8.5.

Unordered Multi-valued Feature Conflict

Conflict detection for unordered, multi-valued features is handled at lines 16 to 29 in Algorithm 7. Instead of using function getUnequalLeftAndRightValues, it employs function getXOR*AndOriginalValues. This 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 unordered features. The procedure to detect a conflict is similar to the procedure for ordered features. The difference is that, to determine whether a conflict is PSEUDO, it checks the existence of values using functions is*Existed.

8.5 Accuracy of Conflict Detection

Conflicts detected by EMF CBP, EMF Compare, and EMF Store can be different because of the different approaches they use. In this section, we explain in more detail the differences between EMF CBP and EMF Compare and then between EMF CBP and EMF Store, concerning the conflicts they can and cannot detect. We use this classification of detected/undetected conflicts later in the evaluation to compare the accuracy of these tools.

8.5.1 EMF CBP vs. EMF Compare

EMF Compare uses model differencing to derive changes—not the real changes—between two versions of a model. This can cause EMF Compare to treat an element or feature as if it has been modified even though in the real context no change has been applied to it. This can lead EMF Compare to inaccurate conflict detection. On the other hand, EMF CBP uses real recorded change events to determine conflicts, so its

conflict detection is accurate. Following are the kinds of conflicts that EMF CBP detects but EMF Compare fails to detect.

- *Real Move Conflict.* EMF CBP accurately identifies an element that has been moved, but EMF Compare picks another element. This case is presented in the running example where EMF CBP detects that target has been moved on both sides (Conflict CB2, Table 8.5), while EMF Compare detects that target and gem have been moved on the left and right sides respectively (Listings 8.1 and 8.2).

- *One-sided Reset Conflict.* EMF CBP detects a PSEUDO conflict on an element or feature that is simultaneously modified but then is set back to its original state on one side (see Figure 8.1d). However, this condition is not determined to be in conflict by EMF Compare since the states of the element or feature are the same in both the original and modified versions.

- *Single-valued Containment Conflict.* The change of state of a single-valued containment feature. EMF CBP detects two different changes to be in conflict if they modify a single-valued containment feature concurrently. For example, element e1 contained in c1.value, and element e2 contained in c2.value, are moved into c3.value concurrently, where value is a single-valued containment feature. Both changes are detected in conflict by EMF CBP but strangely not detected in conflict by EMF Compare.

The following is the only kind of conflict detected by EMF Compare but not detected by EMF CBP.

- *Derived Move Conflict.* This conflict is the opposite of the Real Move conflict. It occurs because EMF CBP records only real moves, not the derived moves produced by EMF Compare. Thus, EMF CBP cannot detect conflicts produced by derived moves.

8.5.2 EMF CBP vs. EMF Store

Even though both EMF CBP and EMF Store real records of changes to determine conflicts, EMF Store does not consider the eventual states of elements or features.

This leads them to detect different conflicts. Following is the only kind of conflict detected by EMF CBP but not detected by EMF Store.

- *First-time Move Conflict.* EMF Store can identify a conflict between two different changes that modify an element concurrently in a multi-valued feature only if both changes are the first changes applied to that multi-valued feature. If an earlier change is applied to another element in the same multi-valued feature, then the following two changes on the same element do cause a conflict. For example, in the original version, a multi-valued feature c1.children contains elements e1, e2, and e3. If in the left version, e2 is moved to the first position and, in the right version, e2 is moved to the last position, then these concurrent changes are detected in conflict by EMF Store. However, if in the left version, the feature is modified with another change, let's say the addition of element e4 at any position, the two move changes are **not** detected in conflict by EMF Store. EMF CBP still detects both move changes in conflict.

Following is the only kind of conflict detected by EMF Store but not detected by EMF CBP. In other words, this should not be detected as a conflict by EMF Store.

- *Two-sided Reset Conflict.* This kind of conflict arises when two sets of changes modify an element or feature but reset its state to the original state on both sides. For example, in the left version, the value of attribute e1.isEnabled is set from false to true, but then it is set back again to false. In the right version, the same changes are also applied to the same attribute. Thus, e1.isEnabled has eventual value false on both versions, the same as in the original version. This kind of change is treated as a conflict by EMF Store but **not** a conflict by EMF CBP (see Figure 8.1c). The same rule also applies to an element that has been moved but then is moved back to its initial position.

Numbers of conflicts detected by EMF CBP and EMF Store can also be different because of the way that EMF Store groups dependent conflicts. For example, let's say that we have a model with initial state element e1 contained in feature c1.value and two other empty features, c2.value and c3.value. On the left side, e1 is moved twice; first to c2.value and then to c3.value. The model is also modified on the right side; a new element e2 is assigned to c2.value, and then another new element e3 is

assigned to c3.value.

In this scenario, EMF CBP identifies two conflicts. The first conflict is a PSEUDO conflict (see Figure 8.1d). That is, c2.value is concurrently modified on both sides, but, on one side, the value is set back to its original state. On the right side, e2 is assigned to c2.value, but, on the left side, c2.value becomes empty when e1 moves to c3.value. The second conflict is a REAL conflict, since c3.value is concurrently modified and has different values on the two sides. On the left side, it contains e1, but, on the right side, it contains e3. EMF Store also identifies these conflicts, but they are merged into one conflict. Another example of conflict grouping can be found in Tables 8.1, 8.2, and 8.5. Conflicts EC3 and EC4 in EMF Compare or conflicts CB3 and CB4 in EMF CBP are grouped into one conflict ES4 in EMF Store since both are in the same composite event I2.

8.6 Evaluation Method

This section presents the method that was used to evaluate the change-based conflict detection approach proposed in this study, and it discusses the results. To assess the performance benefits of the proposed conflict detection approach, this study evaluated it against a mature and widely used state-based model comparison tool, EMF Compare [30, 70], and another implementation of change-based model persistence, EMF Store [11].

Since there are no large, manually developed models persisted in the proposed change-based format yet, the dataset for this experiment was constructed from a large model reverse-engineered from the Eclipse Epsilon project [64, 65]. This model conforms to the Java meta-model [76]. It comprises more than 500 thousand elements with a size of 71.1 MB when persisted in XMI. We aimed for larger sizes of models, but, because EMF Store was slow when it replayed change events, we used the current sizes as they are large enough to identify the performance gaps between the approaches.

The original model was cloned to produce two new (left and right) models, and operations (add, remove, move, set with random elements, features, indexes, and values) were performed 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 was made. The modified models were persisted in state-based format (to be used later in state-based model comparison), and changes persisted in EMF CBP were also replayed on EMF Store to produce equivalent changes. After that, 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 were analysed to capture their trends.

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 reflecting the assumption that, in a mature model, move and set events occur 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 remains 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 assess whether operations of different types have a different impact on model comparison. Because EMF Store is slow when it replays delete events, for the delete-only experiment, the size of the models was reduced from 0.54 million to only 39.5 thousand elements each, and the number of changes was reduced from 0.44 million to 33 thousand 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.

To evaluate the accuracy of conflict detection by EMF CBP, EMF Compare, and EMF Store, we took the change events and states of models produced at the last measurement point of the mixed-operation experiment, and we used them to analyse the conflicts detected by the three tools, based on the classification in Section 8.5.

All measurements were performed on the same machine and software with the following specification: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40 GHz (56 processors), 528 GB main memory, Ubuntu 16.04.6 LTS operating system, OpenJDK Runtime Environment (build 1.8.0_222-8u222-b10-1ubuntu2 16.04.york0-b10) with JVM InitialHeapSize 2 GB and MaxHeapSize 32 GB, EMF Store 1.9.0, EMF Compare 3.3.2, MoDisco 1.0.1, and EMF 2.12.0.

8.7 Evaluation Results and Discussion

This section reports and discuss the results obtained from the evaluation in terms of execution time and memory footprint of EMF CBP, EMF Compare, and EMF Store in detecting conflicts.

8.7.1 Mixed Operations

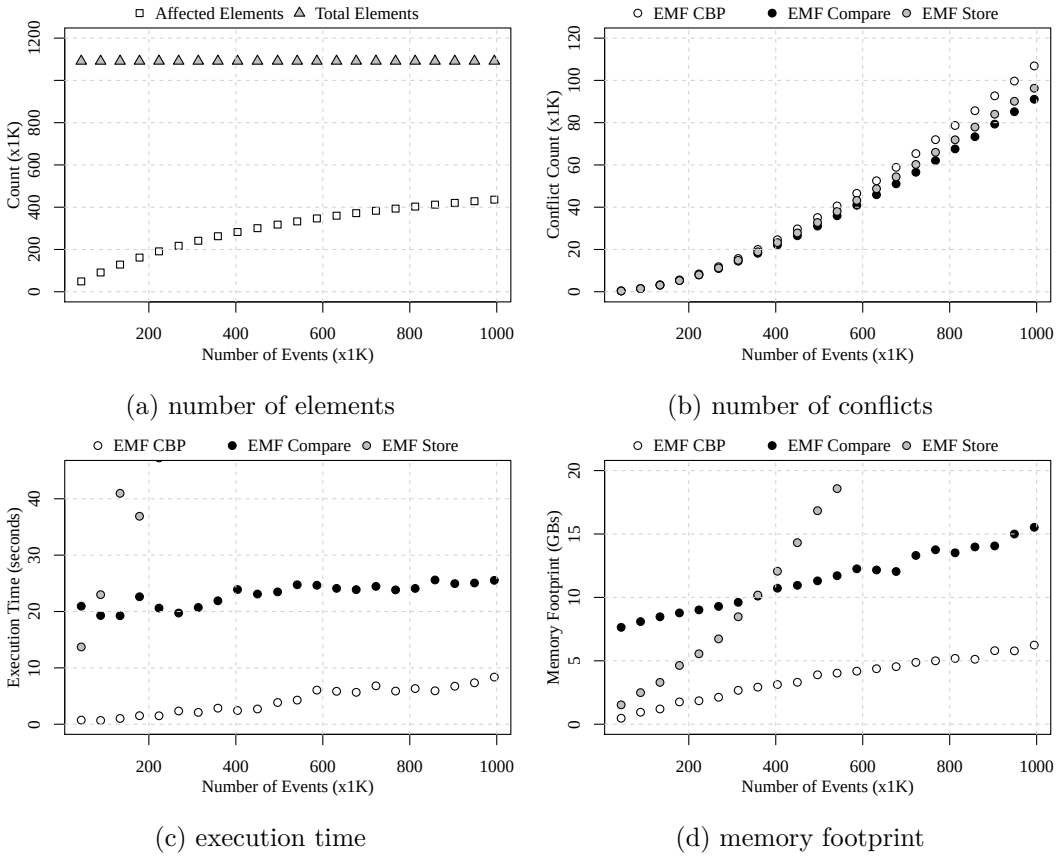
In the mixed operation measurement, we modify two identical models differently by applying random operations. As the number of change events generated by the modification grows, the numbers of affected elements and differences also increase in a logarithmic manner. The patterns are shown in Figure 8.2a. 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 8.2a, the total elements remains largely unchanged because of the equal probabilities of addition and deletion as has been set in Section 8.6. 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.

The growing number of change events in the conflict detection evaluation is followed by the logarithmic increase of affected elements (Figure 8.2a). The total number of these elements can also be kept relatively constant because of 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 shown in Figure 8.2b. The differences are due to their distinct conflict detection approaches. EMF Compare detects fewer conflicts than EMF CBP and EMF Store since its change events are derived, not real changes. EMF Store detects fewer conflicts than EMF CBP since conflicts that depend on each other are grouped into one conflict.

Figure 8.2c shows that EMF CBP outperforms EMF Compare and EMF Store in terms of execution time in detecting conflicts, even when the number of change events approaches one million. EMF Store is the slowest. It takes more than 35 seconds even though the number of change events has reached only 0.1 million. Figure 8.2d also shows that EMF CBP outperforms EMF Compare and EMF Store in terms of memory footprint in conflict detection. At the last measurement point, a million change events, EMF CBP consumes only 6 GB, which is much less than EMF Compare and EMF Store. EMF Compare occupies around 16 GB while EMF Store consumes around 16 GB after only 0.5 million change events.

Figures 8.3, 8.5, and 8.7 show detailed views of EMF CBP, EMF Compare, and EMF Store on the time required for conflict detection. As shown in Figure 8.3, the time for EMF CBP to load change events, construct the element tree, and detect conflicts grows linearly. In detecting conflicts, EMF CBP does not perform differencing since changes are already available in the form of change events. Thus, differencing is not included in that diagram.



(a) number of elements

(b) number of conflicts

(c) execution time

(d) memory footprint

Figure 8.2: Changes in EMF CBP, EMF Compare, and EMF Store as change events increase.

EMF Compare (Figure 8.5), requires less than 5 seconds for matching, and it uses around 15 seconds on average to identify differences. Differencing takes a great portion of the time since it needs to derive differences twice; differences between the left and the original model and between the right and the original model. The time for matching and differencing tends to be constant since the sizes of the models are set to be as constant as possible (Figure 8.2a). 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 the most time to 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 8.7).

In terms of memory footprint, EMF CBP allocates most of the memory space for element tree construction; the rest is for the loading change events and identifying

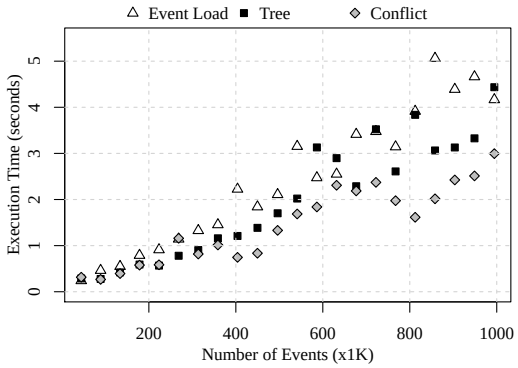


Figure 8.3: Detailed view of EMF CBP on the time required for conflict detection.

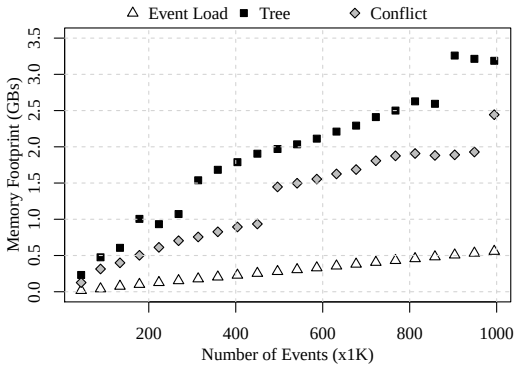


Figure 8.4: Detailed view of EMF CBP on the memory footprint for conflict detection.

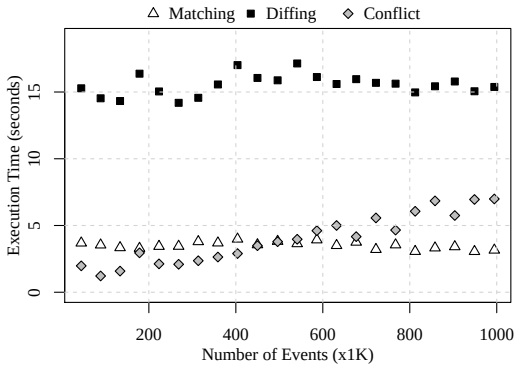


Figure 8.5: Detailed view of EMF Compare on the time required for conflict detection.

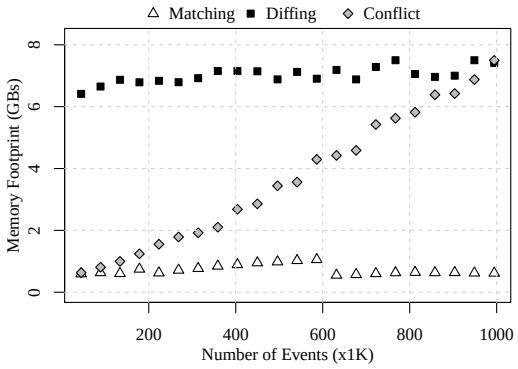


Figure 8.6: Detailed view of EMF Compare on the memory footprint for conflict detection.

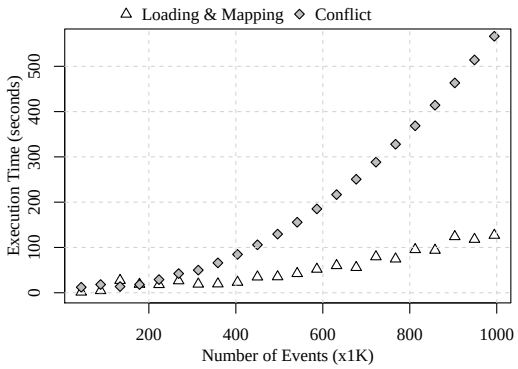


Figure 8.7: Detailed view of EMF Store on the time required for conflict detection.

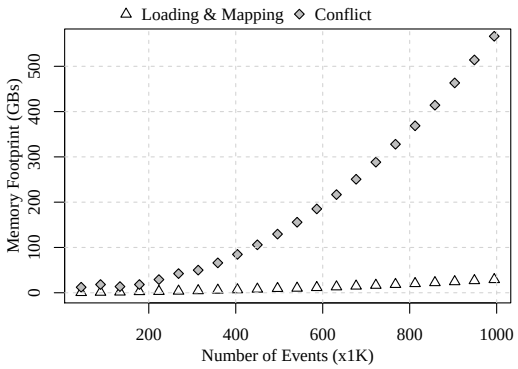


Figure 8.8: Detailed view of EMF Store on the memory footprint for conflict detection.

conflicts (Figure 8.4). The reason for this is our technical implementation in constructing 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 meta-model so that a feature can have only one instance and the instance is used as a key to access the feature's values in each element. This is similar to the implementation of features in EMF Framework. In EMF Compare (Figure 8.6)), the amount of memory used for matching and differencing increases only slightly because the sizes of the models are set to be as constant as possible (Figure 8.2a). In contrast, the memory used for detecting conflict increases as the number of detected conflicts rises (Figure 8.2b). 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 8.8).

From the last measure point of the mixed-operation experiment, EMF Compare detects around 91 thousand conflicts. Around 3 thousand (3.3%) cannot be detected by EMF CBP. This is because EMF Compare derives move changes, which are different from the real changes recorded by EMF CBP. For its part, EMF CBP detects around 107 thousand conflicts, and EMF Compare cannot detect around 19 thousand (18%) of them. These include 6.6 thousand (6.6%) real move conflicts, 8.2 thousand (7.6%) one-sided reset conflicts, and 4.1 thousand (3.8%) single-valued containment conflicts (see Section 8.5.1 to find the definitions of these kinds of conflicts). Thus, there are 88 thousand (91 - 3 = 107 - 19 thousand) conflicts that can be detected by both.

From 107 thousand conflicts detected by EMF CBP, there are 3.7 million (3.5%) conflicts that cannot be detected by EMF Store because of its difficulty detecting first-time move conflicts (see Section 8.5.2). By contrast, EMF CBP cannot detect 1.8 thousand (1.8%) of the 96.4 thousand conflicts detected by EMF Store because of EMF CPB's difficulty detecting two-sided reset conflicts (see Section 8.5.2).

8.7.2 Homogeneous Operations

Detection Time

Figure 8.9 depicts the results of conflict detection time between EMF CBP, EMF Compare, and EMF Store in Homogeneous operations. The results show that, for all

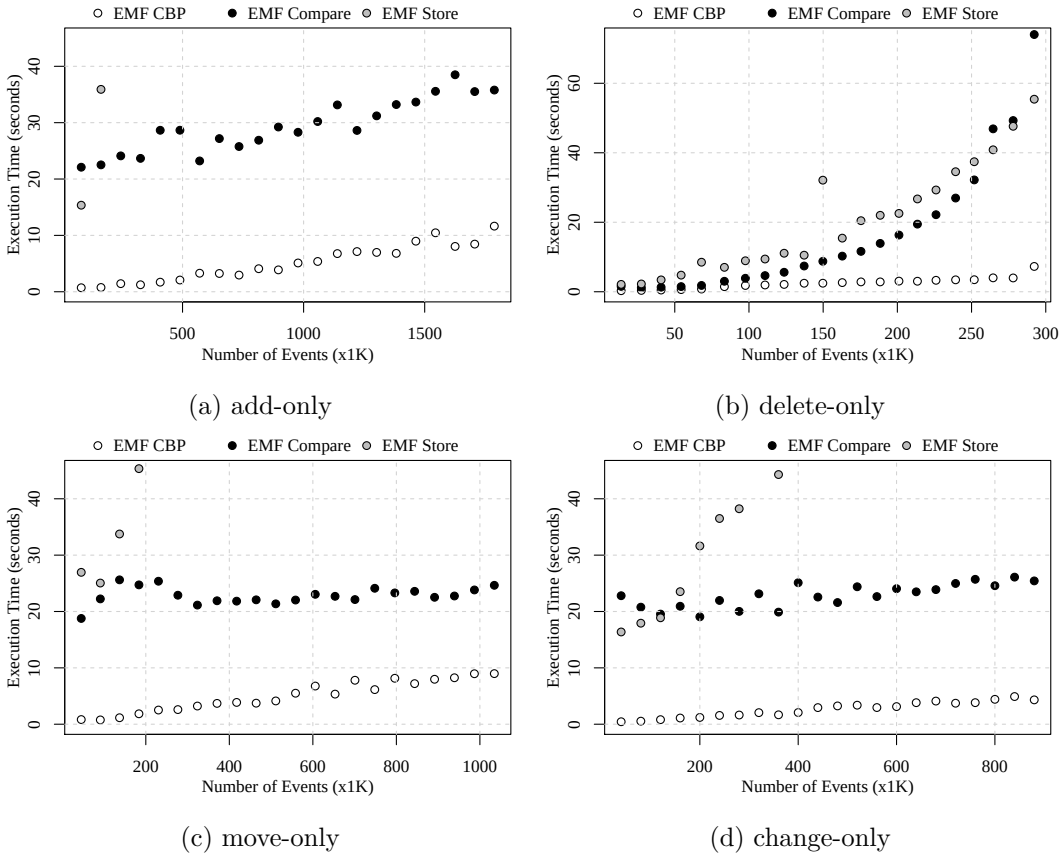
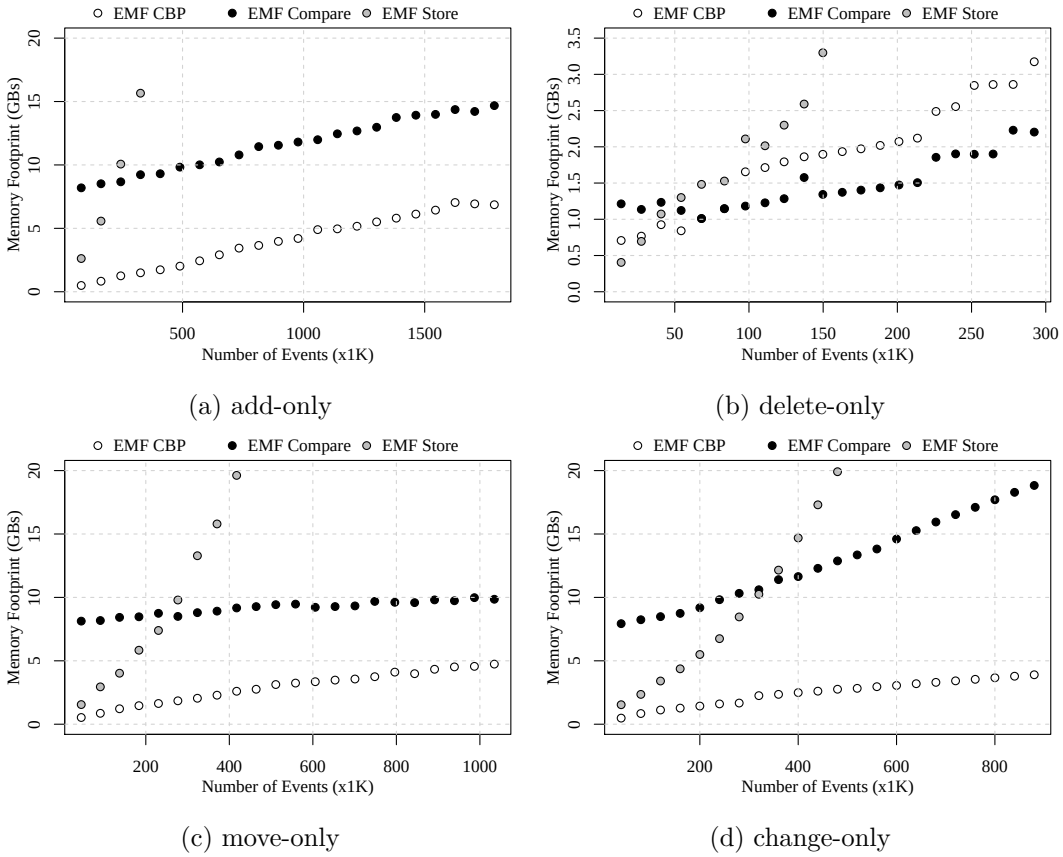


Figure 8.9: Conflict detection time for homogeneous operations.

types of Homogeneous operations, EMF CBP is faster at detecting conflicts than EMF Compare and EMF Store. EMF Store has the worst performance in most cases except for the delete-only experiment. In that case, EMF Compare is the slowest. EMF Compare also requires calculating dependencies between conflicts. So, when the number of deletions is excessive, EMF Compare performs less efficiently than EMF Store (Figure 8.9b). In the evaluation, this happens when the number of change events exceeds 240 thousand.

Memory Footprint

Figure 8.10 illustrates the memory footprint resulting from conflict detection in EMF CBP, EMF Compare, and EMF Store with homogeneous operations. The Figure shows that EMF CBP outperforms EMF Compare and EMF Store in terms of memory footprint. EMF CBP performs worse than EMF Compare only in the delete-only experiment when the number of change events is more than 80 thousand—model size is 39.5 thousand elements each (Figure 8.10b). In terms of memory footprint,



(a) add-only

(b) delete-only

(c) move-only

(d) change-only

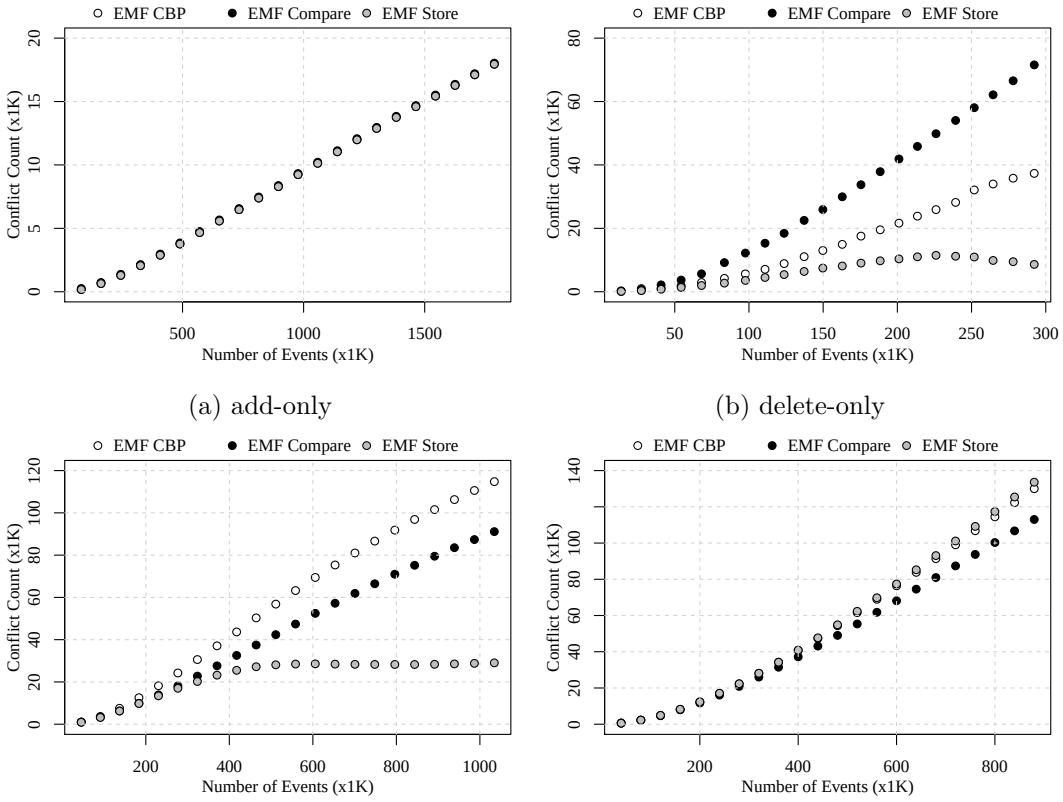
Figure 8.10: Conflict detection memory for homogeneous operations.

EMF Store performs worse than EMF CBP and EMF Compare. It performs better than EMF Compare only when the number of change events is relatively small—fewer than 25 thousand change events.

In Figure 8.10c, EMF CBP's memory footprint increases faster than EMF Compare's memory footprint. This is possible since the change events of EMF Compare are actually minimal differences that are derived from model differencing, which are fewer than real change events recorded in EMF CBP. More random change events means a higher likelihood that more conflicts will occur.

Conflict Count

Figure 8.11 displays the number of conflicts, both REAL and PSEUDO, detected by EMF CBP, EMF Compare, and EMF Store in the context of Homogeneous operations. In the add-only experiment as displayed in Figure 8.11a, all of them detect the same number of conflicts.



(a) add-only

(b) delete-only

(c) move-only

(d) change-only

Figure 8.11: Conflict detection count for homogeneous operations.

Figure 8.11d shows the results of the change-only experiment. We can see that the number of conflicts detected by EMF Compare is lower than EMF CBP. This is mainly because EMF Compare detects no change on an element or feature that has been modified but is changed back to its original state. In EMF CBP, that is counted as a change with potential to raise a PSEUDO conflict as defined and showed in (8.14), Section 8.5.1, and Figure 8.1d. At the last measurement point in Figure 8.11d, there are 17 thousand conflicts of this kind that EMF Compare does not detect. (This is 13.1% of the 130 thousand conflicts that EMF CBP detects) EMF Compare itself detects only 113 thousand conflicts.

It should also be noticed that the number of conflicts detected by EMF CBP is slightly less than those detected by EMF Store. This happens because, as previously discussed, EMF Store does not consider states in detecting conflicts. Thus two different change events that are applied to the same element or feature, even though they yield states that are equal to their original state, are considered to be in conflict.

In Figure 8.11d, at the last measurement point, EMF Store detects 133 thousand conflicts, but 3.1 thousand (2.3%) cannot be detected by EMF CBP because of the two-sided reset conflict (see Section 8.5.2).

In the results for the delete-only experiment in Figure 8.11b, 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 EMF Store does (see Section 8.5.2). 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 to decrease (see Section 8.5.2). In addition, EMF CBP detects fewer conflicts than EMF Compare since it does not calculate conflicts for features and values of an element that have been deleted. Change events that affect features and values are included when calculating conflicts caused by deleting an element, as explained in the last paragraph of Section 8.4.3. In contrast, EMF Compare treats the conflicts at the features and values of a deleted element as separate conflicts.

Figure 8.11c shows the results of the move-only experiment. EMF CBP detects more conflicts than EMF Compare. It has more change events than EMF Compare because of the use of real records of changes. In EMF Compare, change events are derived and effective, which means a minimum number of change events are produced. Fewer change events means there is less likelihood of conflicts. EMF Store detects fewer conflicts than EMF CBP and EMF Compare because of the grouping of conflicts that depend on each other, as discussed in Section 8.5.2.

Using the data on conflicts from the last measurement point in Figure 8.11c, we see that, from 91.6 thousand conflicts detected by EMF Compare, 4.7 thousand (5.1%) are derived move conflicts which cannot be detected by EMF CBP. By contrast, from 114.8 thousand conflicts detected by EMF CBP, there are 27.9 million (24.3%) conflicts cannot be detected by EMF Compare. These include 20.3 thousand (17.7%) real move conflicts and 7.6 thousand (6.6%) single-valued containment conflicts (see Section 8.5.1). We also see that, of the 115 thousand conflicts detected by EMF CBP, 17 thousand (14.8%) are undetected by EMF Store because of the first-time move conflict, explained in Section 8.5.2. On the other hand, of the 29.5 thousand conflicts detected by EMF Store, only 2.5 thousand (8.5%) cannot be detected by

EMF CBP because of the two-sided reset conflict presented in Section 8.5.2.

8.7.3 Conclusions

In this chapter, we have presented an approach to speed up model conflict detection by exploiting the nature of change-based persistence, which allows us to find conflicts between versions of a model by comparing only the last set of changes in the two versions. Based on the findings in the 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 that experience a significant reduction in model size could impair the performance of this conflict detection approach because a great number of change records must be read and loaded into memory.

This chapter has addressed the third research question of this study, **Following change-based model differencing, how can conflicts be detected between versions of a model, and how does change-based conflict detection perform, in terms of speed and memory, compared to state-based model conflict detection?** (RQ3). Similar to change-based model differencing, this work also has proposed an approach to model conflict detection by exploiting the nature of change-based persistence. This allows us to detect conflicts between two versions of a model by comparing only the eventual states of elements and features of the two versions, including their shared original version, that are affected by change events.

The phases in change-based conflict detection are similar to the phases (event loading, element tree construction, and diff computation) in change-based model differencing except that the diff computation is replaced with conflict computation. It also consists of a set of rules that compare 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. As an example, a feature that is modified in only one version cannot have conflicts. A conflict occurs only if the feature is modified in both versions. Also, since the element tree also records every change event to the elements of features that it affects, we can trace change events that cause a conflict.

Chapter 9

Conclusions and Future Work

This chapter summarises the research that we have conducted and the results gained from the evaluation. It starts with conclusions that answer the research questions and hypothesis proposed in Section 3.2. It then presents the limitations and threats to the validity of this research and some topics for future work. Finally, this chapter presents the big picture of this research's contribution to other parts of model-driven engineering, such as model transformation, validation, and evolution.

9.1 Research Questions Addressed

1. **How can models be persisted in a change-based format, and how does change-based persistence perform, compared to state-based persistence, in terms of loading and saving models? (RQ1)**

   This research question is addressed in Chapters 4, 5, and 6. To persist models in change-based format, a prototype was developed. It captures relevant notifications produced by the notification facilities provided by EMF every time a change is applied to an EMF model. It then transforms the notifications into different classes of change events representing different types of changes (e.g., set, unset, add, remove, move, create, and delete) that conform to the model and meta-model infrastructure of EMF. Every captured change event is then persisted by appending it to an XML-like-formatted file when the model is saved. The model can be (re)loaded by de-serialising the file and (re)executing

all the persisted change events—replaying the historical construction of the model.

Since change-based models come with a drawback that their changes must be replayed in order to load them, this work investigated two approaches to improve loading. The first approach optimises loading by not replaying change events that are superseded by subsequent change events. This approach employs a tree-based data structure that tracks all changes made to a model and calculates all superseded events identified by their line numbers. These line numbers are also persisted into another file—ignoreList file—when the model is saved. So, once the change-based model is reloaded, the loading algorithm already knows which change events—which line numbers—should be skipped. This approach can significantly reduce the loading time of change-based models compared to non-optimised loading. However, it is still greatly outperformed by loading models from their state-based persistence, and it suffers greatly in terms of the memory footprint because of the dedicated data structure used to track change events.

In contrast, saving models in change-based persistence shows more favourable results than saving models in state-based persistence, since we need to persist only recent changes in a model rather than saving the entire model. This is very favourable when working with large models at a mature stage where only small changes occur.

Since the results of the first approach are not satisfying, this work also proposed hybrid model persistence—employing change and state-based persistence together. In this type of persistence, models are loaded from their state-based persistence, but changes are persisted into both change and state-based persistence.

In the evaluation, the effects of hybrid model persistence were compared against state-based persistence on loading and saving models in terms of time and memory footprint. The results show that almost all cases experience a slight slowdown on loading and saving time (hybrid approach's $mean >$ state-based approach's $mean$). However, for almost all hybrid NeoEMF cases, the slowdown

is not significant.

The hybrid approach also produces more memory footprint than the state-based-only approach. In terms of storage space usage, on average, persisting one change event consumes only around 100 bytes. This can be used to estimate the growth of storage space usage. For example, persisting 100 million change events consumes around 10 GB.

2. **In a changed-based format, how can the differences between models be identified, and how does change-based model differencing perform, in terms of speed and memory footprint, compared to state-based model differencing? (RQ2)**

   This research question is addressed in Chapter 7. Change-based persistence can be used to identify differences between two versions of a model. The change-based representation of the two versions contains all the information needed to identify elements that have been modified since their last shared version. In this way, we can localise model differencing to the elements that have been recently modified. In other words, it is not necessary to inspect, match, and diff all the elements. We can use the information to reconstruct the partial states of the two versions and then compare their elements and features using specific rules to identify their differences.

   The change-based model differencing proposed in this research 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 that the loaded change events contains are used to construct an element tree. An element tree 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 to construct such a partial representation since change events are designed to contain adequate information to construct the element tree. A diff computation is then executed to identify the differences using a set of pre-defined rules (i.e., if an element is created in one version it means that the element does not exist

in the other version or in the original version).

The evaluation suggests that the proposed change-based model differencing executes faster than traditional, state-based model differencing. However, change-based model differencing needs to load change events from a change-based persistence into main memory. Thus, it can require more memory than for state-based model differencing. In our evaluation, this occurs when the number of change events exceeds 400,000. However, it is likely that diff and merge operations are performed on lower numbers of changes (smaller deltas) than were tested in this evaluation.

3. **Following change-based model differencing, how can conflicts be detected between versions of a model, and how does change-based conflict detection perform, in terms of speed and memory, compared to state-based model conflict detection? (RQ3)**

   This research question is addressed in Chapter 8. Similar to change-based model differencing in the previous research question (RQ2), this work also proposed an approach to model conflict detection by exploiting the nature of change-based persistence. This allows us to detect conflicts between two versions of a model by comparing only the eventual states of elements and features of the two versions, including their shared original version, that are affected by change events.

   The phases in change-based conflict detection are similar to the phases (event loading, element tree construction, and diff computation) in change-based model differencing except that the diff computation is replaced with conflict computation. It also consists of a set of rules that compare 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. As an example, a feature that is modified in only one version cannot have conflicts. A conflict occurs only if the feature is modified in both versions. Also, since the element tree also records every change event to the elements of features that it affects, we can trace change events that cause a conflict.

   Based on the findings in the conflict detection evaluation, this work 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 that experience a significant reduction in model size could impair the performance of the conflict detection because a great number of change records must be read and loaded into memory.

Based on the answers to the three research questions, this work can finally confirm the hypothesis that, **'a textual change-based model persistence approach can outperform existing model persistence formats in terms of model saving, model differencing, and conflict detection time, with an overhead in terms of model loading time and memory use'**. However, this research is not free from limitations and threats to validity. These are presented next.

9.2 Limitations and Validity

This research has tested the proposed algorithms only on synthesised models which were reverse-engineered from two real-world software projects Epsilon [65] and BPMN2 [63], and a collaboratively developed artefact with a long development history, the article on the United States in Wikipedia [66]. The generated models might not be representative of the complexity and interconnectedness of models in other domains. Diverse characteristics of models in different domains can affect the effectiveness of the algorithms and therefore yield different outcomes. Moreover, the generated models from the reverse engineering are limited to the UML2 [68], Modisco Java [76], and Modisco XML [69] meta-models only. Thus, there is no guarantee the algorithms will perform consistently on models that conform to different meta-models.

Specifically in Chapter 5, the proposed loading optimisation of change-based model persistence supports only ordered and unique features. Support for duplicate values means that removing an item does not necessarily result in the item not being present in the feature value. Additional information must be captured to persist the number of copies and positions of the feature members to generate the ignore list.

For the proposed change-based model differencing and conflict detection in Chapters 7 and 8, this work tried to cover many of the common changes made in EMF models (e.g. performing add/remove/set/move operations on single/multi-valued features,

attribute/reference features, or containment/non-containment references). However, the random modification made in the evaluation might not reflect the evolution of models in the real world. This is challenging as different domains can have their own patterns of model evolution, such as different problems, meta-models, and modellers. So far, the most complex composite changes applied to the random modification are limited to move and delete changes. (A move event consists of remove and add events, while delete event also removes the sub-elements of the deleted element.) More complex composite changes, such as refactoring, have not been evaluated. Also, the random modification does not consider the correctness of the changes since it might validate certain constraints of the models. For example, in Java [76] models, removing a parameter from a function causes errors in the function's body, but it is ignored in the evaluation.

9.3 Future Work

The proposed change-based model persistence also comes with a number of challenges for future work, 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 together—in which models are loaded from state-based persistence. Nevertheless, the proposed approach still requires loading change events to construct an elementTree—Section 7.4.2—to perform model differencing and conflict detection, as discussed in Chapters 7 and 8. The loading can be further optimised to consume less memory and speed up parsing by using a binary or a more compact text format.

The challenge of fast-growing model files has not been addressed in this work. Persisting models in a change-based format means that the size of model files will grow significantly faster the model's evolution than their state-based counterparts. Two approaches can be explored in the future to address the issue: (1) sound change-compression operations (e.g. remove older/unused information) to reduce the size of a model in a controlled way, (2) a compact textual format to 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 useful for model analytics as well. 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 or 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 be only 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 use that 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.

9.4 The Big Picture

Model persistence, differencing, and conflict detection are parts of the big picture of Model-Driven Engineering. Regarding model persistence, one might consider in what scenarios change-based model persistence is preferable to state-based model persistence and *vice versa*.

As our findings suggest, change-based persistence can deliver faster model differencing and conflict detection than state-based persistence. This benefit is achieved in the scenario when sizes of models are large and the number of changes is moderate compared to the size of the model. Thus, it is best to use change-based persistence in the later stages of model development when models are already large and changes are mostly for fine-tuning [57]. In this way, storage overhead, because of the growing size of change-based files, can be minimised.

Change-based persistence can become unacceptable in scenarios where the number of changes is excessive relative to the size of the model. The overhead for loading and processing them to construct partial states of models can make the process slower than performing state-based model differencing or conflict detection. This happens in the early stages of model development when models are still small and changes can be numerous and radical. At these stages, state-based persistence is preferable.

The presence of change-based persistence can benefit incremental model management, such as incremental model validation and transformation. Recent changes of models can be efficiently identified without having to perform a state-based comparison to identify the differences between the current and last version of a model. In this way, we can localise model validation and transformation to elements and features that have changed only since the last version. Moreover, the produced change-based model persistence implementation conforms to the standard EMF interfaces and as such change-based models are readable/writable by EMF-compliant transformation and validation languages and engines such as ETL, EVL, OCL and ATL.

While change-based persistence is intended to record changes to models, as a model grows, its meta-model might also experience modifications. How does change-based persistence handle changes at the meta-model level? For now, we have not addressed this challenge. However, one solution that we can suggest to address this challenge is to introduce a new type of change event to be added to the existing types of change events (e.g., add, move, set, create, add, etc.). The new type of change event would indicate an upgrade/downgrade of the meta-model. Another solution is to add the version ID of the meta-model to every change event. In this way, when loading (replaying) the change events of a model, we know whether we need to make some adjustment to handle the model according to the active meta-model.

In terms of generality, one might ask, 'can change-based model persistence, differencing, and conflict detection be applied to any modelling languages?' As long as the modelling languages conform to the EMF meta-modelling architecture, then these operations can be applied. Nevertheless, there is no support for constraints and composite changes that are specific to a modelling language. That belongs to the future work of this research. One way to do that is by using custom adapters.

For composite changes, such as refactoring, the proposed approach also supports composite change events. This feature allows multiple changes that are part of a single refactoring activity to be put into one composite change event. Thus, a change event that conflicts with a member of a composite change event is also in conflict with the other members of the composite change event.

Still related to the generality of the solutions proposed in this research, another

question to answer is 'can the proposed change-based persistence, model differencing, and conflict detection be applied to other artefacts besides models (e.g., XML documents, spreadsheets)?' As long as we can capture all the necessary changes to reconstruct an artefact, then it is possible. Some editors/tools already provide dedicated SDK tools to add custom functionalities. They usually provide access to some kind of event listener, which captures every event executed in the editor/tool. This functionality can be used to capture changes. Also, the format of the persisted changes also needs to be adapted, so that the persisted changes contain adequate information to reconstruct the partial states of the artefact. Once the partial states have been constructed, we can compare the elements of the partial states of the artefact.

Bibliography

[1] Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. CVSM '09, Washington, DC, USA, IEEE Computer Society (2009) 1–6

[2] Robbes, R., Lanza, M.: A change-based approach to software evolution. Electr. Notes Theor. Comput. Sci. **166** (2007) 93–109

[3] Lippe,
SIGSOFT Symposium on Software Development Environments, Washington, DC, USA, December 9-11, 1992. (1992) 78–87

[4] Ignat, C., Norrie, M.C.: 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)

[5] Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. **28**(5) (2002) 449–462

[6] Ráth, I., Hegedüs, Á., Varró, D.: Derived features for EMF by integrating advanced model queries. In: Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings. (2012) 102–117

[7] Ogunyomi, B., Rose, L.M., Kolovos, D.S.: Property access traces for source

incremental model-to-text transformation. In: Modelling Foundations and Applications - 11th European Conference, ECMFA 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-24, 2015. Proceedings. (2015) 187–202

[8] Eclipse: Eclipse Modeling Framework (EMF). https://www.eclipse.org/modeling/emf/ (2019) Accessed: 2019-11-07.

[9] OMG: Metaobject Facility. http://www.omg.org/mof (2018) Accessed: 2018-02-21.

[10] Koegel, M., Herrmannsdoerfer, M., Li, Y., Helming, J., David, J.: 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) 163–172

[11] Koegel, M., Helming, J.: Emfstore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. (2010) 307–308

[12] Yohannis, A., Kolovos, D., Polack, F.: 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) 430–434

[13] Yohannis, A., Rodriguez, H.H., Polack, F., Kolovos, D.S.: 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) 235–250

[14] Yohannis, A., Rodriguez, H.H., Polack, F., Kolovos, D.: 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) 594–603

[15] Yohannis, A., Rodriguez, H.H., Polack, F., Kolovos, D.: Towards efficient comparison of change-based models. Volume 18. (July 2019) 7:1–21 The 15th European Conference on Modelling Foundations and Applications.

[16] Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., Cabot, J.: Neoemf: A multi-database model persistence framework for very large models. Science of Computer Programming **149** (2017) 9 – 14 Special Issue on MODELS'16.

[17] Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K., von Stockfleth, B.: Model-Driven Software Development: Technology, Engineering, Management. Wiley Software Patterns Series. Wiley (2013)

[18] martinfowler.com: UmlMode. https://martinfowler.com/bliki/UmlMode.html (2003) Accessed: 2019-12-03.

[19] Brambilla, M., Cabot, J., Wimmer, M.: Model-driven Software Engineering in Practice. Synthesis Lectures on Software. Morgan & Claypool (2012)

[20] Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Eclipse Series. Pearson Education (2008)

[21] OMG: About the XML Metadata Interchange Specification Version 2.5.1. http://www.omg.org/spec/XMI (2018) Accessed: 2018-02-21.

[22] Git: About. https://git-scm.com/about (2019) Accessed: 2019-11-11.

[23] Apache: Apache Subversion. https://subversion.apache.org/ (2019) Accessed: 2019-11-11.

[24] Myers, E.W.: An O(ND) difference algorithm and its variations. Algorithmica **1**(2) (1986) 251–266

[25] Wang, Y., DeWitt, D.J., Cai, J..: X-diff: an effective change detection algorithm for xml documents. In: Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405). (March 2003) 519–530

[26] Eclipse: Teneo. http://wiki.eclipse.org/Teneo (2017) Accessed: 2017-10-15.

[27] Hibernate: Hibernate ORM. https://hibernate.org/orm/ (2019) Accessed: 2019-11-03.

[28] Eclipse: EclipseLink. https://www.eclipse.org/eclipselink/ (2019) Accessed: 2019-11-03.

[29] Eclipse: Eclipse CDO The Model Repository. https://www.eclipse.org/cdo/documentation/ (2019) Accessed: 2019-04-02.

[30] EMFCompare: Emf compare developer guide. https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html (2018) Accessed: 2018-11-01.

[31] Eclipse: Package org.eclipse.emf.cdo.compare. https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.emf.cdo.doc%2Fjavadoc%2Forg%2Feclipse%2Femf%2Fcdo%2Fcompare%2Fpackage-summary.html (2019) Accessed: 2019-11-03.

[32] Barmpis, K., Kolovos, D.S.: Evaluation of contemporary graph databases for efficient persistence of large-scale models. Journal of Object Technology **13**(3) (2014) 3: 1–26

[33] Espinazo-Pagán, J., Cuadrado, J.S., Molina, J.G.: 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) 77–92

[34] MongoDB: The database for modern applications. https://www.mongodb.com/ (2019) Accessed: 2019-07-23.

[35] Neo4j: Neo4j Graph Platform - The Leader in Graph Databases. https://neo4j.com/ (2019) Accessed: 2019-11-11.

[36] MapDB: MapDB - MapDB. http://www.mapdb.org/ (2019) Accessed: 2019-11-11.

[37] HBase, A.: Welcome to Apache HBase. https://hbase.apache.org/ (2019) Accessed: 2019-11-11.

[38] EMFStore: What is EMFStore and why should I use it? https://www.eclipse.org/emfstore/ (2019) Accessed: 2019-08-15.

[39] EclipseSource: EMFStore - Versioning, History, and Branching. https://eclipsesource.com/blogs/tutorials/emfstore-versioning-history-and-branching/ (2019) Accessed: 2019-11-11.

[40] Eclipse: How to connect EMFStore with MongoDB. https://www.eclipse.org/forums/index.php/t/628706/ (2019) Accessed: 2019-11-11.

[41] EclipseSource: Getting started with EMFStore. https://eclipsesource.com/blogs/tutorials/getting-started-with-emfstore/ (2019) Accessed: 2019-11-11.

[42] Kolovos, D.S., Rose, L.M., Matragkas, N.D., Paige, R.F., Guerra, E., Cuadrado, J.S., de Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: 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) 2

[43] Jouault, F., Tisi, M.: Towards incremental execution of atl transformations. In Tratt, L., Gogolla, M., eds.: Theory and Practice of Model Transformations, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 123–137

[44] Hunt, J., MacIlroy, M.: An algorithm for differential file comparison. Computing science technical report. Bell Laboratories (1976)

[45] Dictionary, O.: diff. https://www.lexico.com/en/definition/diff (2019) Accessed: 2019-07-23.

[46] Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000. (Sep. 2000) 39–48

[47] svn: svn diff (di. http://svnbook.red-bean.com/en/1.8/svn.ref.svn.c.diff.html (2019) Accessed: 2019-07-23.

[48] git: Git - git-diff Documentation. https://git-scm.com/docs/git-diff (2019) Accessed: 2019-07-23.

[49] Consortium, W.W.W.: Extensible Markup Language (XML). https://www.w3.org/XML/ (2019) Accessed: 2019-07-24.

[50] Eclipse: EMF Diff/Merge (EDM). https://www.eclipse.org/proposals/modeling.emf.edm/ (2019) Accessed: 2019-11-05.

[51] Eclipse: EMF Diff/Merge. https://wiki.eclipse.org/EMF_DiffMerge (2019) Accessed: 2019-12-02.

[52] jaxenter.com: Introducing EMF Diff/Merge – chat with Olivier Constant, project lead. https://jaxenter.com/introducing-emf-diffmerge-chat-with-olivier-constant-project-lead-1062.html (2019) Accessed: 2019-12-02.

[53] Treude, C., Berlik, S., Wenzel, S., Kelter, U.: 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. ESEC-FSE '07, New York, NY, USA, ACM (2007) 295–304

[54] Lin, Y., Gray, J., Jouault, F.: Dsmdiff: a differentiation tool for domain-specific models. European Journal of Information Systems **16**(4) (2007) 349–361

[55] Kolovos, D.S.: Establishing correspondences between models with the epsilon comparison language. In Paige, R.F., Hartman, A., Rensink, A., eds.: Model Driven Architecture - Foundations and Applications, Berlin, Heidelberg, Springer Berlin Heidelberg (2009) 146–157

[56] Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: 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) 336–398

[57] Selic, B.: The pragmatics of model-driven development. IEEE Software **20**(5) (Sep. 2003) 19–25

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

[59] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer (2012)

[60] Basili, V.R., Rombach, H.D.: The tame project: towards improvement-oriented software environments. IEEE Transactions on Software Engineering **14**(6) (June 1988) 758–773

[61] EpsilonLabs: emf-cbp. https://github.com/epsilonlabs/emf-cbp (2019) Accessed: 2019-06-06.

[62] Eclipse: MDT/BPMN2. http://wiki.eclipse.org/MDT/BPMN2 (2018) Accessed: 2018-01-15.

[63] Eclipse: BPMN2 Git. http://git.eclipse.org/c/bpmn2/org.eclipse.bpmn2.git/diff/?id=d93667e7418b98acd39def9ed4c2a94501e38997 (2018) Accessed: 2018-02-19.

[64] Eclipse: Epsilon. https://www.eclipse.org/epsilon/ (2018) Accessed: 2018-02-12.

[65] Eclipse: Epsilon Git. http://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/commit/?id=ebd0991c279a1f0df1acb529367d2ace5254fe87 (2018) Accessed: 2018-02-19.

[66] Wikipedia: United States. https://en.wikipedia.org/w/index.php?title=United_States&oldid=45118452 (2018) Accessed: 2018-02-19.

[67] Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: A model driven reverse engineering framework. Information & Software Technology **56**(8) (2014) 1012–1032

[68] Eclipse: MDT/UML2. http://wiki.eclipse.org/MDT/UML2 (2018) Accessed: 2018-01-15.

[69] Eclipse: XML Metamodel. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.xml.doc%2Fmediawiki%2Fxml_metamodel%2Fuser.html (2018) Accessed: 2018-02-19.

[70] Eclipse: EMF Compare. https://www.eclipse.org/emf/compare/ (2018) Accessed: 2018-01-15.

[71] Welch, B.L.: The generalization of 'student's' problem when several different population variances are involved. Biometrika **34**(1/2) (1947) 28–35

[72] Eclipse: Class EContentAdapater. http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/util/EContentAdapter.html (2018) Accessed: 2018-04-20.

[73] Eclipse: Class ResourceImpl. http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/resource/impl/ResourceImpl.html (2018) Accessed: 2018-04-20.

[74] Atlanmod: Interface PersistentResource. http://www.atlanmod.org/NeoEMF/releases/1.0.2/doc/fr/inria/atlanmod/neoemf/resource/PersistentResource.html (2018) Accessed: 2018-04-20.

[75] McKnight, P.E., Najab, J. In: Mann-Whitney U Test. American Cancer Society (2010) 1–1

[76] Eclipse: Java Metamodel. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.modisco.java.doc%2Fmediawiki%2Fjava_metamodel%2Fuser.html (2019) Accessed: 2019-01-08.

[77] Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., Helming, J.: Operation-based conflict detection. In: Proceedings of the 1st International Workshop on

Model Comparison in Practice. IWMCP '10, New York, NY, USA, ACM (2010) 21–30

[78] GitHub: emf.compare. https://github.com/eclipse/emf.compare/ (2019) Accessed: 2019-11-25.

[79] GitHub: emf.diffmerge.core. https://github.com/eclipse/emf.diffmerge.core/ (2019) Accessed: 2019-11-25.

[80] eclipse: emf-store. https://git.eclipse.org/c/emf-store (2019) Accessed: 2019-08-21.

Appendix A

An Example of Change-based Model Persistence

Listing A.1: Change-based representation of the model in Figure 4.2b.

```
1  <session id="ORIGIN" time="20191230131530917GMT"/>
2  <register epackage="miniuml"/>
3  <create eclass="Model" epackage="miniuml" id="O-0"/>
4  <add-to-resource eclass="Model" position="0"><value eclass="Model" eobject="O
       -0"/></add-to-resource>
5  <set-eattribute eclass="Model" name="name" target="O-0"><value literal="ROOT
       "/></set-eattribute>
6  <create eclass="Class" epackage="miniuml" id="O-1"/>
7  <add-to-resource eclass="Class" position="1"><value eclass="Class" eobject="O
       -1"/></add-to-resource>
8  <set-eattribute eclass="Class" name="name" target="O-1"><value literal="
       Character"/></set-eattribute>
9  <create eclass="Operation" epackage="miniuml" id="O-2"/>
10 <add-to-resource eclass="Operation" position="2"><value eclass="Operation"
       eobject="O-2"/></add-to-resource>
11 <set-eattribute eclass="Operation" name="name" target="O-2"><value literal="
       attack"/></set-eattribute>
12 <create eclass="Parameter" epackage="miniuml" id="O-3"/>
13 <add-to-resource eclass="Parameter" position="3"><value eclass="Parameter"
       eobject="O-3"/></add-to-resource>
14 <set-eattribute eclass="Parameter" name="name" target="O-3"><value literal="
       gem"/></set-eattribute>
15 <create eclass="Parameter" epackage="miniuml" id="O-4"/>
16 <add-to-resource eclass="Parameter" position="4"><value eclass="Parameter"
       eobject="O-4"/></add-to-resource>
```

```
17 <set-eattribute eclass="Parameter" name="name" target="O-4"><value literal="
       target"/></set-eattribute>
18 <create eclass="Parameter" epackage="miniuml" id="O-5"/>
19 <add-to-resource eclass="Parameter" position="5"><value eclass="Parameter"
       eobject="O-5"/></add-to-resource>
20 <set-eattribute eclass="Parameter" name="name" target="O-5"><value literal="
       weapon"/></set-eattribute>
21 <remove-from-resource composite="_EuXu4Cr-EeqlN5gavj_cGQ" eclass="Operation"
       position="2"><value eclass="Operation" eobject="O-2"/></remove-from-
       resource>
22 <add-to-ereference composite="_EuXu4Cr-EeqlN5gavj_cGQ" eclass="Class" name="
       operations" position="0" target="O-1"><value eclass="Operation" eobject="O
       -2"/></add-to-ereference>
23 <remove-from-resource composite="_EuXu4Sr-EeqlN5gavj_cGQ" eclass="Parameter"
       position="2"><value eclass="Parameter" eobject="O-3"/></remove-from-
       resource>
24 <add-to-ereference composite="_EuXu4Sr-EeqlN5gavj_cGQ" eclass="Operation" name
       ="parameters" position="0" target="O-2"><value eclass="Parameter" eobject
       ="O-3"/></add-to-ereference>
25 <remove-from-resource composite="_EuXu4ir-EeqlN5gavj_cGQ" eclass="Parameter"
       position="2"><value eclass="Parameter" eobject="O-4"/></remove-from-
       resource>
26 <add-to-ereference composite="_EuXu4ir-EeqlN5gavj_cGQ" eclass="Operation" name
       ="parameters" position="1" target="O-2"><value eclass="Parameter" eobject
       ="O-4"/></add-to-ereference>
27 <remove-from-resource composite="_EuXu4yr-EeqlN5gavj_cGQ" eclass="Parameter"
       position="2"><value eclass="Parameter" eobject="O-5"/></remove-from-
       resource>
28 <add-to-ereference composite="_EuXu4yr-EeqlN5gavj_cGQ" eclass="Operation" name
       ="parameters" position="2" target="O-2"><value eclass="Parameter" eobject
       ="O-5"/></add-to-ereference>
29 <create eclass="Class" epackage="miniuml" id="O-6"/>
30 <add-to-resource eclass="Class" position="2"><value eclass="Class" eobject="O
       -6"/></add-to-resource>
31 <set-eattribute eclass="Class" name="name" target="O-6"><value literal="Troll
       "/></set-eattribute>
32 <create eclass="Class" epackage="miniuml" id="O-7"/>
33 <add-to-resource eclass="Class" position="3"><value eclass="Class" eobject="O
       -7"/></add-to-resource>
34 <set-eattribute eclass="Class" name="name" target="O-7"><value literal="Giant
       "/></set-eattribute>
35 <create eclass="Operation" epackage="miniuml" id="O-8"/>
36 <add-to-resource eclass="Operation" position="4"><value eclass="Operation"
       eobject="O-8"/></add-to-resource>
37 <set-eattribute eclass="Operation" name="name" target="O-8"><value literal="
       cast"/></set-eattribute>
```

```
38 <remove-from-resource composite="_EuXu5Cr-EeqlN5gavj_cGQ" eclass="Operation"
       position="4"><value eclass="Operation" eobject="O-8"/></remove-from-
       resource>
39 <add-to-ereference composite="_EuXu5Cr-EeqlN5gavj_cGQ" eclass="Class" name="
       operations" position="0" target="O-7"><value eclass="Operation" eobject="O
       -8"/></add-to-ereference>
40 <create eclass="Class" epackage="miniuml" id="O-9"/>
41 <add-to-resource eclass="Class" position="4"><value eclass="Class" eobject="O
       -9"/></add-to-resource>
42 <set-eattribute eclass="Class" name="name" target="O-9"><value literal="Knight
       "/></set-eattribute>
43 <create eclass="Operation" epackage="miniuml" id="O-10"/>
44 <add-to-resource eclass="Operation" position="5"><value eclass="Operation"
       eobject="O-10"/></add-to-resource>
45 <set-eattribute eclass="Operation" name="name" target="O-10"><value literal="
       smash"/></set-eattribute>
46 <remove-from-resource composite="_EuXu5Sr-EeqlN5gavj_cGQ" eclass="Operation"
       position="5"><value eclass="Operation" eobject="O-10"/></remove-from-
       resource>
47 <add-to-ereference composite="_EuXu5Sr-EeqlN5gavj_cGQ" eclass="Class" name="
       operations" position="0" target="O-9"><value eclass="Operation" eobject="O
       -10"/></add-to-ereference>
48 <create eclass="Class" epackage="miniuml" id="O-11"/>
49 <add-to-resource eclass="Class" position="5"><value eclass="Class" eobject="O
       -11"/></add-to-resource>
50 <set-eattribute eclass="Class" name="name" target="O-11"><value literal="Mage
       "/></set-eattribute>
51 <remove-from-resource composite="_EuXu5ir-EeqlN5gavj_cGQ" eclass="Class"
       position="1"><value eclass="Class" eobject="O-1"/></remove-from-resource>
52 <add-to-ereference composite="_EuXu5ir-EeqlN5gavj_cGQ" eclass="Model" name="
       classes" position="0" target="O-0"><value eclass="Class" eobject="O-1"/></
       add-to-ereference>
53 <remove-from-resource composite="_EuXu5yr-EeqlN5gavj_cGQ" eclass="Class"
       position="1"><value eclass="Class" eobject="O-6"/></remove-from-resource>
54 <add-to-ereference composite="_EuXu5yr-EeqlN5gavj_cGQ" eclass="Model" name="
       classes" position="1" target="O-0"><value eclass="Class" eobject="O-6"/></
       add-to-ereference>
55 <remove-from-resource composite="_EuXu6Cr-EeqlN5gavj_cGQ" eclass="Class"
       position="1"><value eclass="Class" eobject="O-7"/></remove-from-resource>
56 <add-to-ereference composite="_EuXu6Cr-EeqlN5gavj_cGQ" eclass="Model" name="
       classes" position="2" target="O-0"><value eclass="Class" eobject="O-7"/></
       add-to-ereference>
57 <remove-from-resource composite="_EuXu6Sr-EeqlN5gavj_cGQ" eclass="Class"
       position="1"><value eclass="Class" eobject="O-9"/></remove-from-resource>
58 <add-to-ereference composite="_EuXu6Sr-EeqlN5gavj_cGQ" eclass="Model" name="
       classes" position="3" target="O-0"><value eclass="Class" eobject="O-9"/></
       add-to-ereference>
```

```
59 <remove-from-resource composite="_EuXu6ir-EeqlN5gavj_cGQ" eclass="Class"
       position="1"><value eclass="Class" eobject="O-11"/></remove-from-resource>
60 <add-to-ereference composite="_EuXu6ir-EeqlN5gavj_cGQ" eclass="Model" name="
       classes" position="4" target="O-0"><value eclass="Class" eobject="O
       -11"/></add-to-ereference>
61 <session id="LEFT" time="20191230131531788GMT"/>
62 <create eclass="Generalization" epackage="miniuml" id="L-0"/>
63 <add-to-resource eclass="Generalization" position="1"><value eclass="
       Generalization" eobject="L-0"/></add-to-resource>
64 <set-eattribute eclass="Generalization" name="name" target="L-0"><value
       literal="Left Generalisation"/></set-eattribute>
65 <remove-from-resource composite="_ExFrsCr-EeqlN5gavj_cGQ" eclass="
       Generalization" position="1"><value eclass="Generalization" eobject="L
       -0"/></remove-from-resource>
66 <set-ereference composite="_ExFrsCr-EeqlN5gavj_cGQ" eclass="Class" name="
       generalization" target="O-6"><value eclass="Generalization" eobject="L
       -0"/></set-ereference>
67 <set-ereference eclass="Generalization" name="general" target="L-0"><value
       eclass="Class" eobject="O-1"/></set-ereference>
68 <set-eattribute eclass="Class" name="name" target="O-1"><old-value literal="
       Character"/><value literal="Hero"/></set-eattribute>
69 <unset-ereference composite="_ExFrsSr-EeqlN5gavj_cGQ" eclass="Class" name="
       generalization" target="O-6"><old-value eclass="Generalization" eobject="L
       -0"/></unset-ereference>
70 <set-ereference composite="_ExFrsSr-EeqlN5gavj_cGQ" eclass="Class" name="
       generalization" target="O-9"><value eclass="Generalization" eobject="L
       -0"/></set-ereference>
71 <move-in-ereference eclass="Operation" from="1" name="parameters" target="O-2"
        to="2"><value eclass="Parameter" eobject="O-4"/></move-in-ereference>
72 <unset-eattribute composite="_ExFrsir-EeqlN5gavj_cGQ" eclass="Operation" name
       ="name" target="O-8"><old-value literal="cast"/></unset-eattribute>
73 <remove-from-ereference composite="_ExFrsir-EeqlN5gavj_cGQ" eclass="Class"
       name="operations" position="0" target="O-7"><value eclass="Operation"
       eobject="O-8"/></remove-from-ereference>
74 <delete composite="_ExFrsir-EeqlN5gavj_cGQ" eclass="Operation" epackage="
       miniuml" id="O-8"/>
75 <unset-eattribute composite="_ExFrsir-EeqlN5gavj_cGQ" eclass="Class" name="
       name" target="O-7"><old-value literal="Giant"/></unset-eattribute>
76 <remove-from-ereference composite="_ExFrsir-EeqlN5gavj_cGQ" eclass="Model"
       name="classes" position="2" target="O-0"><value eclass="Class" eobject="O
       -7"/></remove-from-ereference>
77 <delete composite="_ExFrsir-EeqlN5gavj_cGQ" eclass="Class" epackage="miniuml"
       id="O-7"/>
78 <set-eattribute eclass="Class" name="name" target="O-6"><old-value literal="
       Troll"/><value literal="Ogre"/></set-eattribute>
```