Automatic Model Repair Using Machine Learning

Magnus Marthinsen

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2020

Abstract

Model Driven Software Engineering is a field that lets developers focus on the problem they are trying to solve, rather than aiming their attention at implementation details. This is done by creating executable models instead of source code, where the model either is parsed and executed on the fly or used to generate source code. Either way, the models need to be kept correct without any errors. This is a challenging task, as models are worked on by several developers that impose changes on the model. Currently, there exist tools that help developers in dealing with correctness issues. Many of these, however, do not prioritize customization to produce repairs complying with user requirements. Furthermore, they do not allow extension of their components, providing developers with the opportunity to adapt how the algorithm works and extending the reach of what it can handle. This thesis introduces an extensible model repair framework that allows users to customize how the algorithm works by implementing new preferences or adapting the algorithm to handle new model types. The framework uses reinforcement learning to repair models, automatically finding the best sequence of actions that results in a consistent model whilst respecting the preferences declared by the user. This is evaluated by repairing several models with different preference combinations specified through the extensible components, comparing their results. The results from the experiments show that the models are affected by even small changes in the preferences, rendering the user with considerable control over the final repair solution.

Acknowledgements

First and foremost, I would like to thank my supervisors Adrian Rutle, Angela Barriga and Rogardt Heldal, for guidance and helpful advice whilst working on this thesis. Additionally, I would like to thank Ludovico Iovino for his help in implementing the distance function and for his part in calculating quality characteristics of models.

I wish to express my gratitude to my family for the continued support, both academically and otherwise. Come to think of it, mostly otherwise. A special thanks to my father for fostering my interest in computers and technology from an early age, resulting in this degree.

I would also like to thank my room-mate Adrian Storm-Johannessen for not killing me during the Covid-19 pandemic, as we were locked in the apartment together for several months. I would also like to thank him for at least acting interested when I explained the problems I faced in this thesis.

This thesis signals the end of five brilliant years spent at Western Norway University of Applied Sciences, and in that respect, I would like to sincerely thank Kronbar for the important job the organization does for all the students in Bergen. I am extremely grateful for all the wonderful memories I have gathered and lost within the brick walls, and for all the friendships I have made as a result.

Contents

Acronyms **8**

1 Introduction **9**
1.1 Context and Approach . . . 10
1.2 Problem Description . . . 12
1.3 Methodology . . . 13
1.4 Contribution . . . 14
1.5 Outline . . . 15

2 Background **16**
2.1 Model Driven Software Engineering . . . 16
2.1.1 Modeling languages . . . 17
2.1.2 Metamodeling . . . 19
2.1.3 Transformations . . . 19
2.1.4 Automating Development . . . 20
2.1.5 Distance . . . 20
2.2 Model Repair . . . 21
2.2.1 Model Repair Approaches . . . 21
2.2.2 Model Repair Taxonomy . . . 22
2.2.3 Personalizing the repair process . . . 25
2.3 Machine Learning . . . 25
2.3.1 Supervised Learning . . . 26
2.3.2 Unsupervised Learning . . . 27
2.3.3 Reinforcement Learning . . . 27

3 Design and Implementation **30**
3.1 Demonstration . . . 32
3.2 Code origin . . . 36
3.3 Development method . . . 37
3.4 Code structure . . . 38
3.5 Actions . . . 39
3.6 Rewards . . . 40
3.7 Errors . . . 41
3.8 Representation of knowledge . . . 42
3.8.1 Tree structure . . . 43
3.8.2 Map structure . . . 44
3.9 Extensibility . . . 45
3.10 Edit distance . . . 50

4 Use cases **53**

5 Analysis and Assessment **57**
5.1 Does preferences affect final model quality? 58
5.2 Can the preferences be customized? 63
5.3 What is the effect of the model distance? 64
5.4 Can the algorithm handle other model types? 66

6 Discussion **69**
6.1 Analysis results 69
6.2 Infrastructure challenges 71

7 Related Work **73**

8 Conclusion **76**

9 Further Work **78**

A Source code **80**

B Error explanations **81**

List of Figures

1.1 A model having issues with the properties of the variables xMLNSPrefixMap and the xSISchemaLocation. 11
1.2 A screenshot of the Parmorel plug-in being applied to a broken model. 11

2.1 A generic GCS metamodel adopted from [17]. 18
2.2 A simple Web Modeling Language fragment modeling a conference management system in GCS and TCS adopted from [17]. . . 18
2.3 Illustration of the relationships *conform to* and *instance of* adopted from [17]. 19
2.4 Model repair features adopted from [36]. 23
2.5 An example function fitted to training points by a regression algorithm adopted from [3]. The input is simplified to only take mileage, and a linear model is used. 26
2.6 The agent interacts with the environment. The new state of the environment and the reward for the action is returned to the agent. 27

3.1 An overview of the episodes, steps and solutions in Parmorel. . . 31
3.2 A model with three errors. 32
3.3 The settings used in the demonstration. 32
3.4 A model with the errors solved by the algorithm. 36
3.5 The package structure of the algorithm annotated with I for interfaces, A for abstract classes and E for enums. The arrows show inheritance. Note that not all preferences are included in this illustration. 38
3.6 Visualization of a tree structure. 43
3.7 Visualization of a `HashMap` structure. 44
3.8 Parmorel components. 46
3.9 An overview of where in the repairing process the different preference-methods are called. 49
3.10 An overview of the model org.eclipse.wst.ws.internal.model.v10.taxonomy.ecore and its residing errors. 51

4.1 A model containing two errors. 53
4.2 An example of a preference selection. 54
4.3 A list of potential solutions. 54
4.4 Three screenshots of the actions residing in each of the possible solutions present in the example. 55

4.5 A screenshot comparing one of the proposed solutions (left) with the original (right). . . . 55

5.1 A graph showing the difference in maintainability between the results of different settings. The different preference IDs are explained in table 5.2. . . . 60
5.2 A graph showing the difference in understandability between the results of different settings. The different preference IDs are explained in table 5.2. . . . 61
5.3 A graph showing the difference in complexity between the results of different settings. The different preference IDs are explained in table 5.2. . . . 62
5.4 A graph showing the difference in the reuse of the metamodel between the results of different settings. The different preference IDs are explained in table 5.2. . . . 63
5.5 A graph showing the difference in the relaxation index of the metamodel between the results of different settings. The different preference IDs are explained in table 5.2. . . . 64
5.6 A graph showing the difference in execution time between the results of different settings. The different preference IDs are explained in table 5.2. . . . 65

B.1 Error 11 identified in xwt09_updating.ecore . . . 81
B.2 Error 13 identified in activityDiagram.ecore . . . 82
B.3 Error 14 identified in primer.ecore . . . 83
B.4 Error 17 identified in diagramrt.ecore . . . 84
B.5 Error 22 identified in OPF31.ecore . . . 85
B.6 Error 29 identified in car.ecore . . . 85
B.7 Error 32 identified in GSML.ecore . . . 86
B.8 Error 38 identified in OPF31.ecore . . . 87
B.9 Error 44 identified in GUIdancerComponentHierarchy.ecore . . . 88
B.10 Error 48 identified in tableur_modifie.ecore . . . 88
B.11 Error 50 identified in org.eclipse.wst.ws.internal.model.v10.taxonomy.ecore 89
B.12 Error 51 identified in diagramrt.ecore . . . 90

List of Tables

3.1 The empty Q-table for error 1. 33
3.2 The empty Q-table for error 2. 33
3.3 The empty Q-table for error 3. 33
3.4 Q-table for error 1 in the fifth episode. 34
3.5 Q-table for error 2 in the fifth episode. 35
3.6 Q-table for error 3 in the fifth episode. 35
3.7 The empty Q-table for error 4. 35
3.8 The distance resulting from actions applied to the model displayed in fig. 3.10. 52

5.1 Explenation of the acronyms used in the quality characteristic formulas. 59
5.2 Explanation of the preference ID's. 61
5.3 Model maintainability optimized with custom distance calculator. 66
5.4 Model complexity optimized with custom distance calculator. . . 67
5.5 Model relaxation index optimized with custom distance calculator. 67
5.6 Percentage of models that improved, worsened or remained unchanged with respect to the quality characteristics after applying custom distance. 68

Acronyms

API Application Programming Interface.

EMF Eclipse Modeling Framework.

GCS Graphical Concrete Syntaxes.

IDE Integrated Development Environment.

MDE Model Driven Engineering.

MDSE Model Driven Software Engineering.

ML Machine Learning.

PDE Plug-in Development Environment.

RL Reinforcement Learning.

RQ Research Question.

SE Software Engineering.

TCS Textual Concrete Syntaxes.

TDD Test-Driven Development.

UML Unified Modeling Language.

XMI XML Metadata Interchange.

XML Extensible Markup Language.

Chapter 1

Introduction

Software Engineering (SE) is an engineering discipline that focuses on the development of high-quality software systems [60]. One of the challenges in SE is the high complexity in the systems created [18, 50]. The systems are very difficult to implement correctly, sometimes consisting of millions of lines of code. Furthermore, the core of all programming is formal logic, which is either correct or incorrect. A single flaw can invalidate an argument.

Another challenge in SE is that software is invisible and cannot be visualized [18]. In other fields, geometric abstractions can be very powerful. An example is the floor plan of a building that helps both the client and the architect evaluate the spaces. A geometric abstraction represents the geometric reality and help communicate ideas and avoid contradictions. Software, however, does not embed itself in physical space. Therefore, no geometric representation works like it does for maps over land or connectivity schematics for computers. When software is represented in a diagram, it results in several overlapping directed graphs. They may represent flow of control, flow of data or time sequence to mention some system aspects. These graphs are usually neither planar nor hierarchical. Because software is unvisualizable the brain cannot use some of its most powerful conceptual tools, which not only hampers the design process but cripple communications with others.

Model Driven Software Engineering (MDSE) is an engineering discipline that aims at dealing with these challenges. The core concepts in the MDSE context are *models* and *transformations* [17]. Models are abstractions and can be used to reflect something and transformations are operations that manipulates models. Models are important for understanding and communicating complex software, and MDSE applies the advantages of modeling to SE activities.

Modelling is an important activity in the field of SE [62]. To maintain a good quality during development it is important to keep the models correct and accurate [8]. The challenge of keeping the models free of errors grows with the number of changes made to the models and the size of the teams working on them. Keeping track of all the versions of the same model and confirming its validity is a difficult task.

This thesis will aim to reduce this burden for developers by proposing a tool extending the algorithm created in [8, 9, 10] that can automate some of the process. If successful, the time spent on manually dealing with correctness issues diminishes. Such tools have the potential of improving delivery time, the final quality of the developed product and greatly improve how organizations deals with Model Driven Engineering processes [8]. The tool proposed in this thesis utilizes Reinforcement Learning (RL), a form of Machine Learning (ML). Because the repair process is automatic with personal preferences, the tool is called Parmorel (**P**ersonalized and **A**utomatic **R**epair of **Mo**dels using **Re**inforcement **L**earning)[1].

1.1 Context and Approach

When selecting the modeling framework to work with, we wanted our approach to be as unrestricted as possible. Hence, the code is built in such a way that it can be extended to support multiple modeling frameworks. We chose the Eclipse Modeling Framework (EMF) as the first framework to be supported by the plug-in because EMF is widely used in the modeling community [56].

EMF is the core technology in Eclipse for model driven engineering [17]. It allows the definition of metamodels (section 2.1.2) based on *Ecore*, a metamodeling language. Ecore is based on a subset of UML class diagrams for describing structural aspects and is tailored to Java for implementation purposes. Ecore is itself an EMF model, and can be viewed as a meta-metamodel [56].

EMF provides generator components that can produce a Java-based API from the metamodels [17]. This provides a way of manipulating the models programmatically. The generator components can also produce modeling editors to build models in tree-based editors from the metamodels. EMF comes with a powerful API that covers various aspects such as serializing and deserializing models to and from XMI in addition to powerful reflection techniques. Several other projects are based on EMF and provides further functionality for building model-based development support within Eclipse.

Considering EMF exploits facilities provided by Eclipse, the plug-in is developed using the Eclipse Plug-in Development Environment (PDE). We chose to implement the algorithm in Java in order for it to be easily integrated in the plug-in and work well with EMF.

The resulting plug-in can trigger a repair process on a broken model file (e.g. fig. 1.1) as shown in figure 1.2. If the file is an ecore-file, the Parmorel repair option appears. When selected, the user is prompted with a list of preferences on which to base the repair. After the repair process has finished the user is presented with several solution alternatives, ordered by how the solutions matches the user's preferences. In order to understand the various alternatives, the user can view the actions taken in each solution, and graphically compare it to the original. If the user likes a solution, it can be selected as a repair replacing the original model.

[1] `https://ict.hvl.no/project-parmorel/`

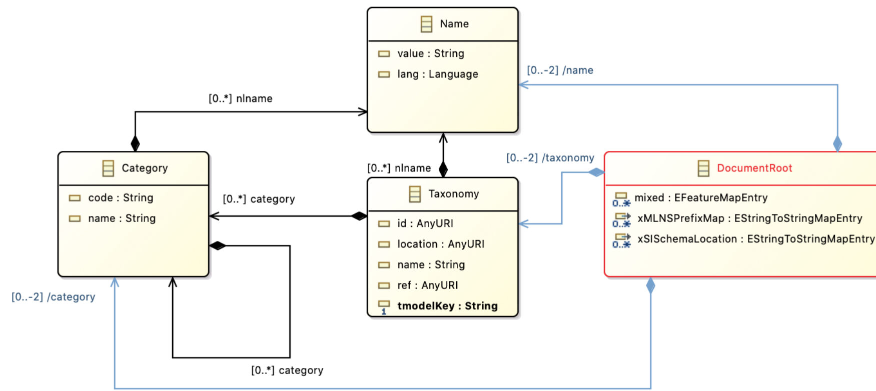

Figure 1.1: A model having issues with the properties of the variables xMLNSPrefixMap and the xSISchemaLocation.

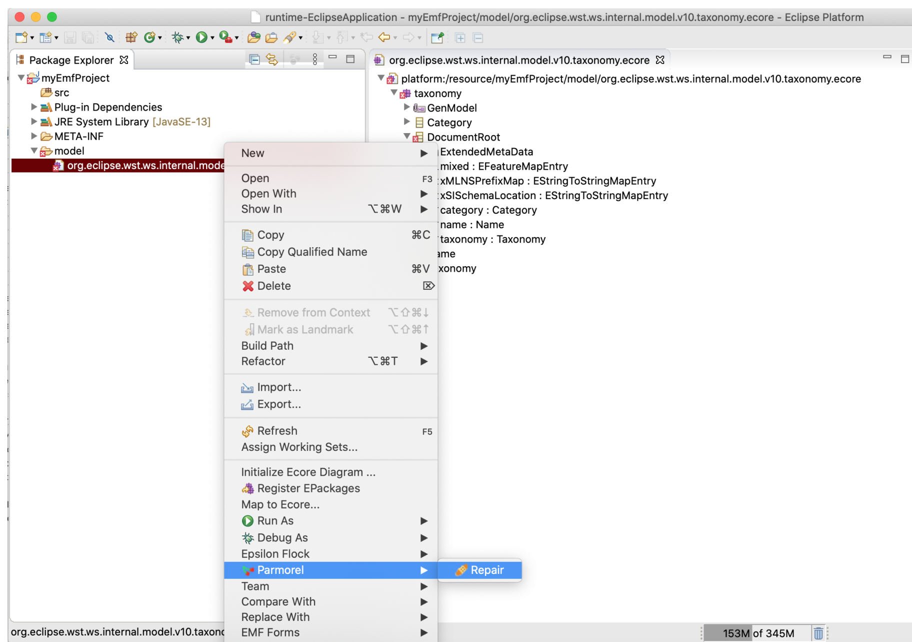

Figure 1.2: A screenshot of the Parmorel plug-in being applied to a broken model.

[36] defines a unifying taxonomy for model repair approaches explained in section 2.2.2. Most of the domain-related features and the constraint features are not relevant for Parmorel, due to the fact that Parmorel leaves the reporting of errors and the finding of available actions to the modeling framework. The Parmorel algorithm is metamodel independent but needs some extending for each metamodel before it supports it (explained in section 3.9). This is also the case for the *technical space*, where the current implementation only supports EMF.

Parmorel can be decoupled from the modeling framework, because the algorithm is state-based. It uses a decoupled checking procedure that must return a set of violations occurring in the model along with some information about the errors. The information must include a unique identifier for the type of error and a location specifying the package and the context in the model where the error resides.

The algorithms core is *search-based*, but also *incremental* as it reuses the Q-table to improve efficiency over time. The repair is represented both *state-based*, i.e. as a new generated model instance, and *operation-based*, i.e. as a repair plan listing the actions that have been applied to the model in order to solve the errors.

1.2 Problem Description

In this thesis a useful tool for developers is proposed, providing the ease of automatic bug fixing when performing modeling activities. The tool is a continuation of the work done in [8, 9, 10] proving RL feasible when repairing models. The proposed tool is inspired by quick fix solutions already implemented in Eclipse, highlighting errors and provides actions that can be undertaken to repair them. The proposed tool differs by working on models rather than just coding and compiling errors for source code.

The aim was to achieve human performance in fixing models by applying ML for autonomous model repair. When working with model repair and ML, a few challenges arise. One of them is the lack of datasets available publicly in the field of modeling [8]. Most ML algorithms requires big labeled datasets referred to as *training data* in order to achieve good results (explained in section 2.3). RL, however, does not require training data and thus it becomes useful for domains where the historical data is inadequate. We chose Q-learning, explained in section 2.3.3, as our RL-algorithm because of its table structure. By storing the knowledge in a generic format, it is easy to export and import the data between executions.

When performing automatic repair on a model, the search space can grow exponentially dependent on the content of the model [8]. The search space is the set of all feasible outcomes [13], and there might exist an overwhelming number of updates to resolve any given set of inconsistencies [36]. Complexity and uncertainty are added to the algorithm as a result of the variety in possible solutions. Additionally, potential fixes are generally not unique.

Automated repairing techniques has a weakness in providing the same solutions for certain errors. Yet, developers possess different preferences for repairing models [10]. Automated techniques must find a balance between the user guidance when generating the alternative solutions and the level of automation [36].

Hence, we have identified the following research questions:

- **RQ1** - To what degree does the personal preferences affect the proposed solutions?
- **RQ2** - How can the application be reengineered and what adaptions must be made to:

    A add new preferences to the algorithm?

    B make the algorithm work with other model types?

Concretely, RQ1 refers to how much the user can affect the resulting model through the preferences selected before the repairing process begins. RQ2 asks how the application can be altered and improved. This is detailed in two further questions regarding preferences (RQ2A) and the ability to handle other model types (RQ2B).

1.3 Methodology

This paper is a solution-seeking study [57] that aims to ameliorate the process of repairing models in MDSE. The study is a method of development [51], looking at improving how developers work in MDSE by evolving the model repair algorithm proposed by [8, 9, 10]. To improve how developers work with models we will attempt to build a plug-in and test how much the users' choices affect the repaired models. We will also explore different ways of providing the algorithm with the users' preferences and how the algorithm can be evolved to handle other model types.

There is also an element of feasibility study in this thesis [29, 38], because it explores whether or not it is possible and practical to create a tool for developers to use whilst constructing and altering models. However, it misses the business aspect of a feasibility study in that it disregards financial components and focus solely on the technical aspects.

The success of this thesis will be evaluated on how much the user can affect the outcome of the repair procedure, and how extensible the algorithm is. If the algorithm has successfully been implemented in Eclipse and a user can achieve different solutions from the model repair procedure using different preferences the project will be considered successful. If the process can be even more tailored to the users by adding custom preferences this is a major bonus. Lastly, the approach should be generic meaning models other than Ecore can be supported. This is not vital to the assignment but is an interesting way of finding the algorithms limits. The project has failed if the tool does not offer the user with any ways of affecting the repair procedure, or if it is not able to repair models at all. The algorithm will be tested on broken models. The datasets containing these models are introduced in chapter 5.

1.4 Contribution

As mentioned, this thesis did not start from scratch, but expands upon the work done in [8, 9, 10]. To this end, Q-learning had been proven to work for EMF model repair by creating an algorithm combining RL-automation with personal preferences before the start of this thesis.

> Clean code can be read, and enhanced by a developer other than its original author [37].

This was unfortunately not the case for the original code. It was built as proof-of-concept, consisting of few classes containing many lines of code. As a result, it was hard to understand and almost impossible to expand upon. A lot of work has gone into refactoring the original code to follow modern programming practices and creating a simple API towards the algorithm. This will, hopefully, enable the algorithm to be maintained over time and make it possible to add new functionality to it.

Adding functionality has also been a major part of this thesis. One major change is that the algorithm is no longer tied to EMF but has become an extensible framework that can be extended to support other model types. This was the primary focus during the refactoring process.

The original algorithm provided seven preference options to the user. In addition to adding another preference based on the *distance* (explained in section 3.10) to the original model, the preference implementation has become customizable as part of the framework. This results in other developers now being able to create custom preferences and add them to the algorithm.

How the algorithm gathers experience has also been altered in this thesis. Instead of just relying on the preferences, the end user has been given the option to select a solution and reward it. This will make the algorithm prefer solutions similar to the chosen ones over time.

The code has also been documented with Java-docs, making the code more understandable to other developers.

The Eclipse-plug-in is also a contribution from this thesis, providing a graphical user interface for repairing models in the development environment.

In addition, the thesis assesses the approach and conducts a set of experiments to evaluate the impact of personal preferences on the final model and the success of the preference extensibility.

These contributions made to enhance the extensibility of Parmorel and its assessments are submitted as a scientific paper [11] to the technical track of the Models 2020 conference[2]. The Models 2020 conference, MODELS, the ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, is the premier conference series for model-driven software and systems engineering and is organized with the support of ACM SIGSOFT and IEEE TCSE.

[2] `https://conf.researchr.org/track/models-2020/models-2020-technical-track`

1.5 Outline

This thesis is structured with the theoretical background that is important to know in order understand the main aspects of this study in chapter 2, and the technical design and implementation of the produced repair tool presented in section 3. Chapter 4 presents various anticipated use cases for the tool. Different aspects of the algorithm are analyzed by experiments in chapter 5, with the results discussed in chapter 6 along with other aspects of the thesis. Related work is presented and compared to the work conducted in this thesis in chapter 7 before the conclusion is presented in chapter 8. Finally, chapter 9 presents aspects not explored in this thesis that should be conducted in the future.

Chapter 2

Background

In this chapter, we will present some of the knowledge that our research is built upon. This theory is important to know in order understand the following chapters.

2.1 Model Driven Software Engineering

In this section Model Driven Software Engineering (MDSE) is explained as described by [16, 17]. It can be defined as a methodology for implementing the advantages of modeling in engineering activities. The core concepts in the MDSE context are *models* and *transformations*. Models are abstractions and can be used to reflect some system by using two features.

1. They only reflect a relevant section of the original's properties. This is called a *reduction* feature.
2. Models are based on some original individual that is abstracted and generalized to a model representing a category of individuals. This is called a *mapping* feature.

Transformations are operations that manipulates models. Models are the primary development artifact in MDE.

An example of a model from a scientific context is the Bohr model of the atom. This model is probably an unacceptable simplification, but it has been outstandingly helpful in understanding the basics of chemistry and physics. A transformation on an atom could be to combine it with another atom creating a molecule.

MDSE aims at dealing with increasingly complex software by focusing on the *problem domain* instead of on the *solution domain*. The problem domain is defined as the environment that needs to be studied in order to understand and define a problem. A conceptual model of this domain is called a *domain model*. This model describes different entities in the environment by specifying their attributes, roles and relationships. The model also describes the constraints and

interactions regarding the entities that make up the problem domain, granting integrity to the model. The solution domain is the choices made at design, implementation and execution level when creating a software application that solves a problem specified within the problem domain.

2.1.1 Modeling languages

Modeling languages are tools for specifying models in graphical or textual representations. As models can be defined both graphically and textually, existing framework currently support two kinds of concrete syntaxes: Graphical Concrete Syntaxes (GCS) and Textual Concrete Syntaxes (TCS). These languages have a formal definition, and the designers must follow their syntax when modeling.

A GCS defines *graphical symbols* for visualizing model elements, e.g. lines, graphic figures and labels for representing textual information. It also defines *compositional rules* which defines how the graphical symbols can be nested and combined. Finally, a *mapping* from the graphical symbols to the elements defined in the abstract syntax is defined, stating which graphical symbol should be used for which modeling concept.

Current graphical modeling editors use a modeling canvas, which allows the model elements to be positioned in a two-dimensional pixel map. The model elements are usually arranged as a graph contained in the modeling canvas. A piece of a generic GCS metamodel (explained in section 2.1.2) is shown in fig. 2.1. The metamodel states that a diagram consists of various elements in the form of nodes, edges, compartments and labels. Nodes and edges are graph concepts represented by shapes and lines, respectively. Compartments are used to nest elements, namely diagrams are *nested graphs*. Additional information can be attached to nodes and edges with labels, i.e., diagrams are also *attributed graphs*.

Textual specifications assume that the text consist of a sequence of characters. A grammar is needed to specify all the valid character sequences, as arbitrary sequences probably will not conform to a valid specification. This grammar needs to be generic to allow the rendering of models textually as well as parsing the text to models. It is also desirable with syntactic sugar like language-specific keywords to enhance the readability. Consequently, a TCS requires some additional concepts.

Model information is one of the common elements that is important for every TCS and must be supported to allow model information to be stored in the abstract syntax. This is achieved through the use of labels in GCS. Another element is *keywords*, i.e., words that has a particular meaning in the language. These words become reserved and cannot be used as values for model elements. *Scope borders* are also needed, as no figure exists to define the border of a model element like in GCS. In TCS this is usually achieved by special dedicated characters to mark the beginning and the end of a certain section. *Separation characters* is another special character used to separate entries in lists, providing the possibility to list elements at certain positions. Lastly, *links* that allow referencing elements related by non-containment references must be supported.

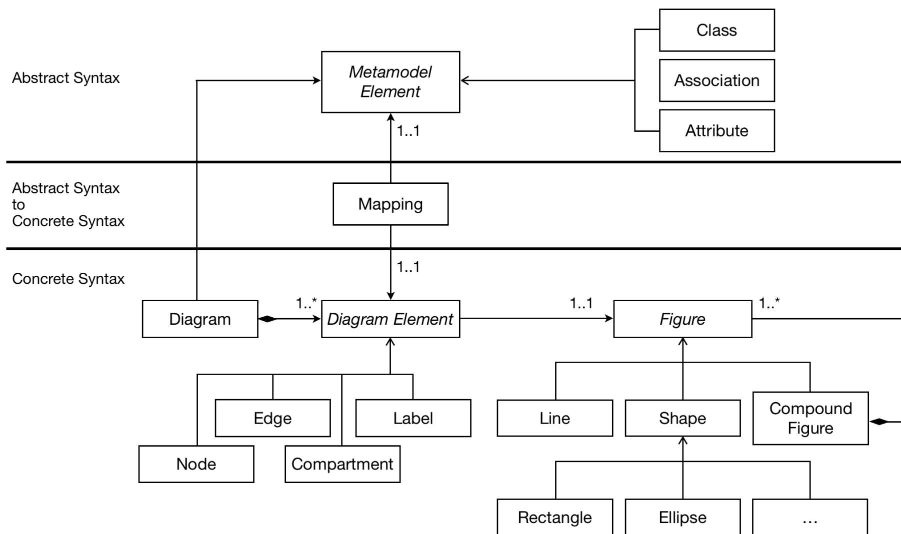

Figure 2.1: A generic GCS metamodel adopted from [17].

This is equivalent to edges in GCS. In TCS however, we only have one dimension in which to define the elements. Instead of edges, identifiers are defined for the element which other elements can use to reference it.

Possible visualizations of a conference management system are shown in figure 2.2 as an example, with a GCS on the left and a TCS on the right. This side-by-side comparison will be used to clarify the elements described above. The model information are things like the name of the model elements as well as their type. The keywords, marked in bold in the TCS, are in this example used to introduce the various model elements. The scope-borders are curly brackets, opening after an element is introduced by its keyword and closing after the element is defined to make up the compartment of the element. The separation characters are semi-colons, separating the different attributes introduced for the class. Lastly, the page must be linked to a class of the content layer. Programming languages are commonly using class names as types, so the name of the class becomes a natural identifier.

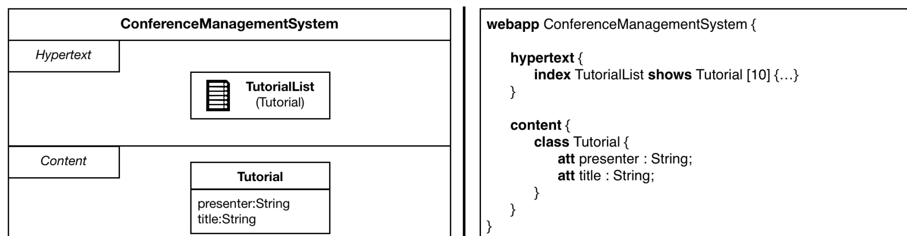

Figure 2.2: A simple Web Modeling Language fragment modeling a conference management system in GCS and TCS adopted from [17].

2.1.2 Metamodeling

Now that the models can be defined by a syntax, the next step is to represent the models themselves as *instances* of more abstract models. In much the same way as models describe something in the real world through abstractions, a *metamodel* can be defined as yet another abstraction emphasizing properties of the models. The definitions of modeling languages can be viewed as metamodels, because they provide a way of describing the whole class of models that can be produced from the language.

In other words, models describe something from the real world. These models can again be described by other models (called metamodels). Additional models describing the metamodels (called meta-metamodels) can be defined recursively. This could go on for infinity, but it has been shown that meta-metamodels can be defined based on themselves. As a result, it does not make sense to abstract more levels than this. If all elements of a model can be expressed as instances of (meta)classes in a corresponding metamodel, it is said that the model *conforms* to the metamodel. This is illustrated in figure 2.3.

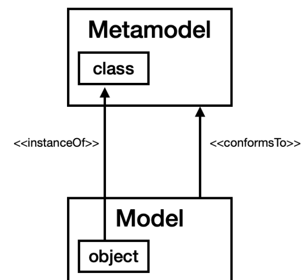

Figure 2.3: Illustration of the relationships *conform to* and *instance of* adopted from [17].

Metamodeling frameworks allow dedicated editors to specify the metamodels and generate modeling editors out of the metamodels. The metamodel can be interpreted as a set of *production rules* for building the model. This can be used to render the model elements in different representations, e.g. a TCS. The metamodels can also be interpreted as a set of *constraints* that is used to verify that a model conforms to the metamodel.

2.1.3 Transformations

Transformations allow mappings between different models to be defined. These transformations are defined at the metamodel level but applied at the model level to models that conform to the metamodels. In other words, the transformation is performed between a source and a target model but is defined upon the respective metamodels.

Designers use appropriate languages for defining model transformations for specifying transformation rules. These languages can be used for defining model transformations through templates that can be applied to models corresponding to some matching rules checked upon the model elements.

Transformation rules can be written by a developer from scratch. Alternatively, existing ones can be redefined for a more specific use case. Transformations can also be produced automatically based on some higher-level mapping rules between models. This technique includes two steps:

1. Define a *model mapping*, mapping elements of a model to elements of another.
2. Use a system that receives the two models and a mapping between them as input and generates the actual transformation rules.

This process lets the developer focus on the conceptual aspects of the model relations and pass on the responsibility for producing the transformation rules.

Transformations on a model can be either *out-place* or *in-place*. Out-place transformations generates the resulting transformed model from scratch. This is well suited for transformations between two different languages, e.g. from a platform independent model like UML to a platform specific model like Java. This is called *exogenous* transformations. In-place transformations rewrites a model by creating, deleting and updating the elements in the model. This is convenient for operations like refactoring.

Models can, as mentioned in section 2.1.1, be expressed as graphs. As a result, the models can be manipulated by graph transformation techniques. *Graph grammars* consist of an initial graph and a set of graph transformation rules to be applied to it. A left-hand side (LHS) graph and a right-hand side (RHS) graph makes up the root of the rule. The pre-conditions are expressed through the LHS, whilst the RHS specifies the post-conditions. Combined, both sides imply what actions will be executed when the rule is applied. All elements exclusive to the LHS will be deleted, all elements that only reside in the RHS will be added and the remaining elements are preserved. The elements require an identifier assigned to them in order to detect equality between elements in the LHS and the RHS.

2.1.4 Automating Development

The models created in MDSE are not only artifacts created to help the development. They are the development. The models are used to generate a running system, and this requires the models to be executable. Two strategies exist for making the models execute.

Code generation works in much the same way as code compilers creating binary files from source code, by generating code from the higher-level model. The "code" generated can be source code, but also test cases, documentation, configuration files etc.

Model interpretation is another approach that does not generate code from a model in order to create a working application. Rather, a generic engine is implemented, parsing and executing the models themselves. This interpretation approach works like interpreters do for interpreted programming languages.

2.1.5 Distance

Models can, as mentioned in section 2.1.1, be viewed as graphs. As a result, graph theory can be applied to the models. This includes *edit distance*, the shortest sequence of edit operations to transform one graph to another [20].

Edit distance is also applicable to text, again the smallest number of operations required to transform one string into another [28].

When calculating differences between models, existing approaches can be divided into three main phases [1]:

1. Compared models are imported in a differencing friendly format, typically graph-based structures.
2. A matching algorithm traverses the models, detecting and establishing correspondences between entities in them.
3. The differences of the matched elements in terms of (at least) additions, deletions and changes of the model entities is computed by a dedicated algorithm based on the matches from the previous phase.

2.2 Model Repair

Model repairing is a research field that can improve how engineers interact with model-driven projects. In this section Model Repair is explained as described by [36].

The models developed with MDE are modified by different stakeholders. This requires a consistently monitored and managed MDE environment. Consequently, various activities involved in the detection, diagnosis, handling and tracking of inconsistencies is essential to MDE.

These inconsistencies might be introduced to the environment due to mistakes or careless decisions as developers apply changes to the models. The impact of these changes might not be obvious right away, taking into account the complexity of the MDE environment. The inconsistencies must be handled eventually, and automated techniques that helps the user restore consistency is a necessity for this to be manageable.

One such technique is to propose update actions to the user that will *repair* the models, hence improving the consistency level of the MDE environment. One of the main challenges in model repair is the potential huge number of updates that might exist for each set of inconsistencies. The decision of the most suitable update to fix the model is ultimately up to the developer. Therefore, approaches to model repair must find a balance between the level of automation and the need for user guidance.

2.2.1 Model Repair Approaches

There are several ways to approach model repair. *Rule-based* model repair is one approach that require a set of previously defined rules that is applied to the model when an error occurs. This provides full control over how to resolve errors but demands more from the designer having to specify how the constrains are fixed. The flexibility of the approach is also restricted to the fixed set of resolution rules.

Generative approaches rely on production rules that define what a well-formed model is. From these rules a set of transformation rules are derived. Generative approaches are suitable to use with graph grammars, where repair rules are derived from the grammar productions.

Syntactic approaches automatically derive the repair plans from a syntactic analysis of the constraints instead of from the production rules that generative approaches use. These repair plans are typically calculated at static-time, and only instantiated to concrete model instances when inconsistencies are found at run-time. Syntactic techniques may be able to derive repair solutions without user input but may result in so many potential solutions that the user will struggle to select one. Additionally, these techniques are ill equipped to handle multiple inconsistencies in one model and inconsistencies that affect a big chunk of the model.

Search-based approaches interpret model repair as a model search problem. In contrast with the syntactic approaches, these procedures are well-suited to handle inconsistencies affecting large parts of the model. Although the approaches are able to automatically find fully consistent models, they have problems with scaling. These techniques differ in how they find consistent states. Some rely on *off-the-shelf solvers* to search for consistent states sometimes resulting in unpredictable solutions due to total disregard of the application domain. Others rely on *domain-specific* search procedures using domain-specific knowledge, like available edit operations, that results in a finer control when generating repair updates.

Some techniques are *hybrid*, meaning they are built over more than one of these features. An example is rule-based approaches that utilizes search-based methods to calculate repair plans from the rules.

When repairing models, different techniques can handle different domain spaces.

2.2.2 Model Repair Taxonomy

A unifying taxonomy for model repair approaches is defined by [36] and describes several feature groups illustrated in figure 2.4. It sums up features making up the *domain*, i.e. which model instances the technique can handle and whether the user is able to customize the domain space. One of these features is *formalism* that explains what model representations the repair technique can handle (e.g. graphs, object-oriented specifications etc.). Another feature is whether or not the approach aims to be independent of the application domain, i.e. *meta-model independent*. If the technique deals with fixed metamodels it is aimed at some particular classes of models, for example UML diagrams. Otherwise, it allows the designer to restrict the model domain space, adapting it to the user's needs. The *technical space* is the space in which the user specifies the various artifacts of the MDE development environment, e.g. XML or EMF. If restrictions are imposed on the model elements, the repair technique is said to be *bounded*. A typical restriction is not allowing the creation of new elements, resulting in the repair being bounded by the elements present in the model. The last feature in the domain is *multi-model*, meaning the technique focuses on handling inter-

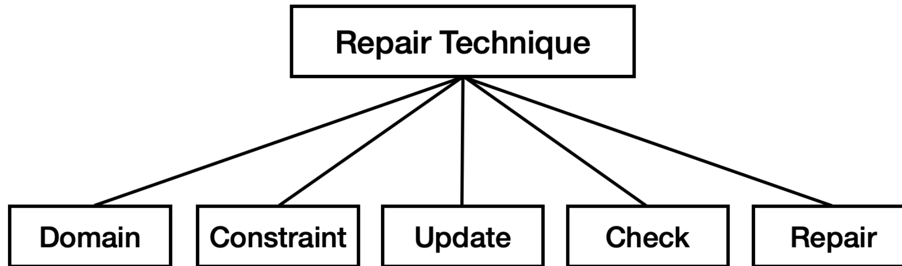

Figure 2.4: Model repair features adopted from [36].

model constraints between several models rather than intra-model consistency in a single model.

Constraints is another feature group defined by the taxonomy. The implementation of the constraints affects the repair procedure, as a violation of a constraint translates to a broken model. Hence, the way the constraints are shaped determines what type of properties the repair procedure can handle. How the constraints are specified and whether or not they are customizable by the user is one feature called *specification*. This feature also includes if the approach supports *repair hints*, meaning that each constraint is supplied with hints from the user on how to repair a violation. Lastly, the specification includes the optional support of *distinguished* constraints, meaning the procedure can focus on a few constraints even if this results in negative effects for the other constraints. The feature *kind* defines what type of constraints are supported, referring to inter-model or intra-model constraints. The last constraint feature determines the *shape* of the constraints supported by the approach, e.g. logical predicates of pattern matching in a graph.

Different repair techniques have different information available to them regarding the evolution of the model from the previous known state to the current one. This information is addressed by the *update* features. The first feature is *update representation*, which is either *state-based* or *delta-based*. Delta-based approaches have information about the user actions that led to the current state of the environment, whilst state-based only considers the current state of the model. The former call for a dedicated modeling framework, whilst the latter allows the framework to be decoupled from the repair technique. The inconvenience of tracking the actions are rewarded with more predictable repair updates. The second feature in the update category is what *extra information* is available to the algorithm, e.g. the model state before the error was introduced or the history of the model evolution.

Other differences between repairing schemes is the reliance on the validation procedure of the model. This is the procedure that tests if the resulting model violates the provided constraints after an action is applied. The taxonomy calls this feature category *check*. In addition to specifying whether or not the resulting model is valid, it might supply additional information. The check procedure might be *decoupled*, meaning that external tools detects the violations and reports them. Some techniques supply a *checkonly* mode, meaning the inconsistencies in the model is detected and reported, but not repaired. The

reporting of the checking procedure classifies what information is reported about the model. This might be a simple Boolean specifying whether or not the model is valid or report the number of violations in the model. The most common is to return the set of violations detected, containing various information regarding the violations. Such reports can be compared to check the consistency level of the model. It might be more consistent after a repair has been completed, but maybe not fully consistent yet.

Repair is another feature category classifying the overall repair behavior. The feature *core* categorizes the underlying repair generation procedure. This can for example be *rule-based*, applying a previously defined rule whenever an inconsistency is detected. *Search-based* techniques interprets model repair as a model search problem. These can either rely on domain-specific search procedures requiring domain-specific knowledge or be oblivious to the application domain. *Incremental* techniques reuse data from previous executions in order to improve efficiency. The repair category also includes the feature *repair representation*. This feature describe what information is returned by the repair procedure. *State-based* approaches return a new generated model instance, whilst *operation-based* approaches return a repair plan listing the actions that have been applied to the model in order to solve the errors. The *content* of the repair updates is either *concrete*, meaning they can be applied directly to the environment without any user input, or *abstract*, requiring input from the user before they can be instantiated.

Another feature group is *Enumeration* defining how the repair updates are selected and presented to the user, in addition to how this can be controlled. One enumeration feature is *output*, describing how many repair alternatives are returned by the algorithm. This might be a single repair update, or several. If it returns all possible repair updates, it is said to be *complete*. *Order* is the other enumeration feature, describing the ordering of the returned updates. This might be opaque to the user, or predefined making the procedure more predictable. The order might be *parameterizable*, meaning the user has some control over the repair procedure.

The last feature group, *semantics*, explores what semantic properties the repair procedure is guaranteed to follow. *Totality* is a feature explaining how good a procedure is when finding solutions. If the algorithm can produce a repair update for every user update that results in an error (as long as one is available and the modeling framework provides the necessary actions), the technique is said to be *total*. *Correctness* is another feature in semantics. This defines what guarantees the technique provides for correctness. If the inconsistency level of a model does, at least, not increase when a repair update is applied the procedure is *well-behaved*. A technique that guarantees to improve the inconsistency level is *consistency improving*, and if the inconsistency level is guaranteed to be reduced to the minimum it is *fully consistent*. Semantics also include *stability*, saying a procedure is *stable* if it does not return any repair updates if provided with a stable model.

2.2.3 Personalizing the repair process

Regarding the enumeration output, the approaches need to present the user with a manageable number of acceptable updates. This forces the techniques to restrict themselves, given the possible number of repair updates might initially be overwhelming. When restricting the number of presented repair updates, different procedures output even a single repair update or multiple repair alternatives.

Techniques that return every possible repair update within the parameters of the execution is said to be *complete*. The parameters of the execution include the bounds of the search space, allowed edit operations and constraints imposed by the enforced semantic properties. If an approach is not complete, it might discard interesting repair options or fail to handle certain inconsistencies.

The set of returned repair updates and the order in which they are presented to the user must somehow be chosen by the procedure. Users may be allowed to affect the calculation of this order, giving the user more control over the repair procedures behavior. One example of this is a procedure that uses graph-edit distance and lets the user attach weights to the meta-model, hence prioritizing repair update for some model element types over others. Alternatively, the weights can be attached directly to the model elements, promoting changes to concrete parts of the model instances.

Another way of letting the user affect the repair process is to allow control over the edit operations that makes up the repair updates by assigning them with costs. Users might also be able to prioritize the defined constraints, making the procedure focus on specific classes of violations. Lastly, the user might be able to control the procedure through additional meta-data like versioning information.

Interaction is another way for the user to control the algorithm. To facilitate this, the technique uses interactive dialogs to refine the set of possible repair updates.

2.3 Machine Learning

In this section Machine Learning (ML) is explained, including some popular algorithms and different branches of the field.

ML are algorithms learning from the surrounding environment, designed to emulate human intelligence [42]. The learning process uses input to automatically change the algorithms architecture, and through repetition it performs better and better. This repetition is referred to as experience, and the process of altering the architecture is called training.

There are multiple ways in which ML algorithms can learn, and two of them are *supervised learning* and *unsupervised learning* [14]. When an algorithm uses supervised learning, it uses some *training data* to construct a function. The training data is usually a dataset with a label for each entry that specifies the correct answer. The algorithm uses this data to construct a function for

predicting the results. Unsupervised learning does not use labeled data but involves pattern recognition without a label specifying the target attribute [15].

2.3.1 Supervised Learning

Supervised learning is commonly used to estimate a mapping from input to output, based on known sample data [42]. The output is labeled, making this sort of algorithms suitable for tasks like *classification* and *regression*.

In classification, the algorithm takes an input and returns either a specific label, or a number specifying the confidence score for a particular label [21]. An example of a classification task of distinguishing apples from oranges [42]. Even though all apples and oranges are unique, we are still usually able to tell them apart. For this to be supervised learning, we need some training data describing color, shape, odor etc., and a label specifying the correct prediction. A successful learner should be able to recognize previously unseen apples and oranges after training on this data. Some common methods used for classification tasks are decision trees, probabilistic methods and SVM-methods [2].

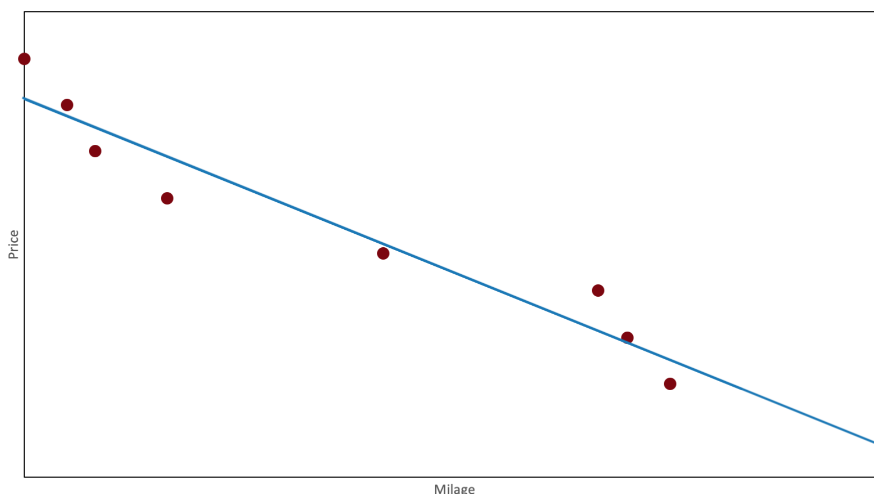

Figure 2.5: An example function fitted to training points by a regression algorithm adopted from [3]. The input is simplified to only take mileage, and a linear model is used.

In regression, the algorithm takes an input and returns a number as output [3]. An example of a regression task is taking in attributes for a used car and returning the current selling price of the car. Examples of input attributes are brand, year and mileage. By examining transactions for used cars, we can collect training data for a regression algorithm. The algorithm will fit a function to the data to predict the output price Y as a function of the input X. An example is visualized in figure 2.5. Some common methods used for regression tasks are Least Squares [21], Logistic Regression [34] and SVM-methods [54].

2.3.2 Unsupervised Learning

Supervised learning finds a mapping from input to output, and the correct output is provided by labels on training data [3]. Unsupervised learning does not have these correct labels, only the input data. The algorithm looks to find some regularity in the input, for example a structure where some patterns occur. These regularities can be used to see what generally happens and what does not. This is referred to as a *density estimation* in statistics.

An example of a common unsupervised learning task is clustering [42]. Clustering is a task for finding clusters/groups within the input data [3]. This is useful for operations like *image compression*, where the objective is to reduce the file size of the image. The input are the pixels that make up the image, and each pixel is represented by an RGB-value. If the image consists of 16 million colors each pixel needs 24 bits to be able to represent all of them. However, if we group similar colors together so all the colors are displayed as shades of 64 main colors only 6 bits per pixel is required. For example, an image with various shades of blue can display the same color for all the pixels. This will reduce the details of the image but save storage space.

Another example of unsupervised learning is throwing a dart at a bull's eye [42]. When throwing the dart, it can be thrown in different angles and with different strength. The learner will practice throwing the dart, and for each throw the angles and strengths are adjusted so the dart gets closer to the bull's eye. This is unsupervised, because the training does not associate an input of angles and strengths with an outcome but finds its own way from the training input data. A successful learner should be able to adjust for changes in the environment, for example a reposition of the target.

2.3.3 Reinforcement Learning

Some algorithms output a sequence of *actions* [3]. In such cases, a single action is not important. What is important is that the sequence of actions together reaches the goal. There is no single best action in the sequence, but an action is good if it is part of a good sequence. The algorithm needs to assess how good a sequence is and be able to learn from good sequences in order to generate new sequences. Such learning methods are called Reinforcement Learning (RL).

In RL the learner is called an agent [3]. The agent performs actions in an environment that results in a change of the environments state. The feedback to the agent is given in the form of reward or punishment, illustrated in figure 2.6. When selecting what action to perform the agent tries to maximize the total reward of the action sequence. It is not required to give feedback to the agent for each action, and sometimes the reward is only given at the end when the complete sequence is produced.

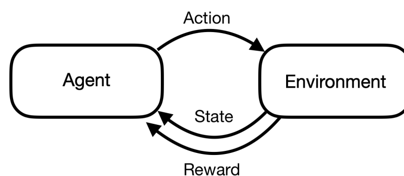

Figure 2.6: The agent interacts with the environment. The new state of the environment and the reward for the action is returned to the agent.

A good application of RL is game playing, where a single move itself is not important [3]. A move is only good if it is part of a sequence of moves that together perform well in the game. If the game is chess, the game-player will be the agent and the board will be the environment.

Another application for RL is a robot in a maze [3]. In this case, the agent is controlling the robot, the environment is the maze and the state of the environment is the robot's position. The agent can move in any of the four compass directions and should make a sequence of moves to eventually reach the exit of the maze. In this case, no feedback is given until the robot reaches the exit. Even though there is no opponent, shorter paths can be preferred. This implies that the agent is playing against time.

Q-learning

In this section an RL algorithm called Q-learning will be explained with regards to [8, 10, 61].

Q-learning is a form of RL. The agent tries an action in the environment at a particular state. It evaluates the consequences of the action based on the immediate reward/penalty it receives *and* its estimate of the value of the new state of the environment as a result of the action. It learns which actions are the best overall by evaluating a long-term discounted reward.

The long-term discounted reward is stored in a structure called Q-table and is continuously updated while the agent interacts with the environment. From a given state the algorithms find the most optimal action by consulting this Q-table. For each action (a_t) it selects a *Q-value* calculated by the Bellman Equation. This Q-value is the maximum future reward (r) the agent received for entering the current state (s_t) with the selected action (a_t), plus the maximum future reward for the next state (s_{t+1}) and action (a_{t+1}). To avoid falling into a local maximum this score is reduced by a discount factor γ. When performing optimizations there might be several viable solutions. Some methods might find the closest viable solution from the starting point (local maximum), even though there might exist a better solution in the environment. The best solution in the environment is a global maximum. The Q-value allow to infer the value of the current state (s_t) based on the calculation of the next state (s_{t+1}), which can be used to calculate an optimal policy for selecting actions.

$$Q(s_t, a_t) = r + \gamma \left(\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \right)$$

The calculated Q-value is used to update the Q-table. A factor α specifies a learning rate, which regulates how much the stored Q-value changes from one iteration to the next.

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left(r + \gamma \left(\max_{a_{t+1}} Q_t(s_{t+1}, a_{t+1}) \right) - Q_t(s_t, a_t) \right)$$

The algorithm can now determine the value of a state and select the optimal action until the final goal is achieved. This process is called *exploitation* and is repeated every time (t) an action is selected by the algorithm.

The algorithm can also pick actions at random, instead of choosing the optimal action from the Q-table. This process is called *exploration* as it explores alternative, possibly more optimal, solutions. By combining the exploitative and explorative approaches a wider range of solutions can be found.

The agent is given several attempts at solving the problem, and one attempt is called an episode. For each episode the agent will try to reach the final goal by performing several steps, performing exploitative or explorative choices trying to find the best action for the current state. For every chosen action the Q-table is updated. The number of episodes and the number of steps per episode is decided with regards to the problem size and should ensure enough time to reach the final goal.

Chapter 3

Design and Implementation

In this chapter the implementation of the algorithm will be explained. First an overview of Parmorel is given including an example, with detailed technical information regarding specific components later in the chapter. Because Parmorel is developed and tested with regards to EMF, the examples will be based on the Ecore metamodel. In section 3.9, an explanation on how the algorithm can be extended to handle models derived from other metamodels is provided.

The repairing algorithm is powered by Q-learning (section 2.3.3), in which the experience gained is stored in a Q-table. In our implementation, the Q-table is three-dimensional. Every entry in the table is a combination of what actions have been applied to which error, and to what context. The entry has a weight that reflects how good an action is for repairing an error at a given location in the model.

The first thing the algorithm does when repairing a model is to find all the errors currently in the model. Then it makes sure all the error codes found in the model exist in the Q-table. If the Q-table does not contain an error code, it is added along with corresponding contexts and actions. The contexts are objects describing the error by being the primary source of the problem, or objects associated with or describing the problem. For example, if two features have the same name, each feature and the class containing them represents a context in which the error can be repaired. The algorithm will search through this Q-table to find actions that can be applied to the model when attempting to repair it. To narrow the search space for the algorithm, actions that do not result in a change in the model are omitted (get-methods etc.) when populating the Q-table. The algorithm checks if the error still exists after the action is applied, and if it does not the action is added to the Q-table for the corresponding action.

After the Q-table is initialized, the Q-learning algorithm starts repairing the model over several episodes. Each episode is an iteration where the algorithm repairs the model within a predefined number of steps. Each step is an application of one action to the model. The action is either the most optimal action from the Q-table chosen by weight, or a random action. A random action might be chosen to find an alternate path, avoiding a local maximum. The maximum number of steps is determined by formulae 3.1. It is set to 1.4 times the number

of errors (NOE) in the model but never below 20. This is to accommodate models with few errors as new errors might be introduced in the repairing process. The algorithm should have enough steps available to reach a consistent model. However, the algorithm might get stuck if it consistently tries to apply the wrong action. This is resolved when the maximum number of steps is reached and the episode is terminated (or by a random action), but this might take a long time if the algorithm has many steps remaining. A balance between a high and low maximum number of steps must be found, and formulae 3.1 have provided good results on the models we have tested.

$$Maximum\ number\ of\ steps = \begin{cases} 1.4 \cdot NOE, & 1.4 \cdot NOE > 20 \\ 20, & 1.4 \cdot NOE \leq 20 \end{cases} \tag{3.1}$$

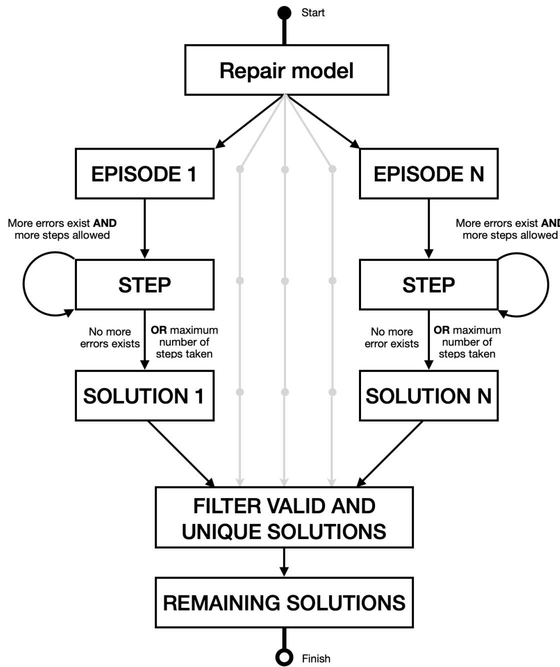

Figure 3.1: An overview of the episodes, steps and solutions in Parmorel.

When the action is applied to the model, the Q-table is initialized for any new errors that potentially has entered the model. The new state of the model is used to calculate the reward for applying the action to the old state. When calculating rewards user preferences are considered, tailoring the repair process to the user. The reward starts as zero and is updated by iterating over all the selected preferences that add or subtract preference-specific numeric values to the reward.

In this implementation, the number of episodes is set to 25 if no previous knowledge is found or 12 otherwise. This may result in the same number of possible solutions, but equal solutions are discarded. By repairing the model many times, the chances for finding the optimal solution increases. The solution the user selects will receive a boost in the Q-table for all the actions performed in order to achieve it.

Figure 3.1 shows an overview of the algorithm regarding how the episodes, steps and solutions are connected. In the first episode several steps are executed until no more errors exist in the model, resulting in a solution. This is repeated several times, and if the resulting solution is valid and unique it is added to the list of possible solutions. Otherwise the solution is filtered out. The knowledge is accumulated over several episodes, meaning that the experience gained in episode 1 is used and evolved in episode 2 and so forth.

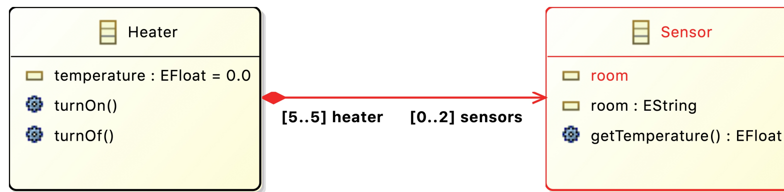

Figure 3.2: A model with three errors.

3.1 Demonstration

In this section some details of the algorithm will be explained with an example. For simplicity, the Q-table is initially empty at the beginning although it is possible to load the Q-table from earlier executions of the algorithm. Note that the algorithm has a random element and might not produce the same result for each execution.

Consider the model shown in figure 3.2. This is an Ecore model describing a central heating system, where a heater has sensors in various rooms. The model has three violations of the constraints imposed by the metamodel. EMF reported the following errors:

- **Error 1**: There may not be two features named 'room'
- **Error 2**: The typed element must have a type
- **Error 3**: A container reference must have upper bound of 1 not 5

The error numbers have been altered for simplification in this example and are not the same numeric values reported by EMF. Error 1 is referring to the Sensor-class, that has two attributes named room. Error 2 is referring to the room-attribute that does not have a type, highlighted in red. Error 3 is referring to the reference between the classes. It specifies that if the heater is to contain the sensors, a sensor can only belong to one heater and not five.

Each of these errors can be solved in different ways. Error 1 can for example be fixed by renaming or deleting one of the attributes or delete the entire `Sensor`-class. Error 2 can be fixed by giving the attribute a type or deleting it. Error 3 can be solved by setting the upper bound to 1, modifying the containment to a regular reference or deleting the whole reference. This small example illustrates the complexity involved in repairing models, no matter how simple they are.

- [ ] Prefer shorter sequences of actions
- [ ] Prefer longer sequences of actions
- [ ] Prefer repairing higher in the hierarchies
- [x] Prefer repairing lower in the hierarchies
- [ ] Prefer modification of the original model
- [ ] Punish modification of the original model
- [x] Punish deletion

Figure 3.3: The settings used in the demonstration.

In the following example, the algorithm is run with the settings shown in figure 3.3.

Context	Action name	Weight
0	delete	0.0
1	setName	0.0
1	delete	0.0
2	setName	0.0
2	delete	0.0

Table 3.1: The empty Q-table for error 1.

Context	Action name	Weight
0	setEType	0.0
0	delete	0.0

Table 3.2: The empty Q-table for error 2.

First, the algorithm extracts the errors from the model and initializes the Q-table with actions for these errors. This will result in a Q-table like table 3.1, 3.2 and 3.3.

The algorithm starts the first episode, consisting of up to 20 steps. The maximum of 20 steps is set by formulae 3.1 based on three initial errors. For each step, the algorithm might take the optimal action or a random action, as explained in section 2.3.3. The algorithm starts by repairing Error 1.

In this demonstration, the algorithm chose to start with a random action `setName` in context 2. It receives a reward of 200 based on the preferences and calculates a new Q-value for entering the next state Error 2.

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left(r + \gamma \left(\max_{a_{t+1}} Q_t(s_{t+1}, a_{t+1}) \right) - Q_t(s_t, a_t) \right)$$

$$Q_{\texttt{Error 2}}(s_{\texttt{Error 1}}, a_{\texttt{setName}}) = 0 + 1 \cdot (200 + 1 \cdot 0 - 0) \tag{3.2}$$

$$= 200 \tag{3.3}$$

In the next step, the algorithm chose an optimal action. Because all the values are zero for this error, the algorithm does not know which action is considered

Context	Action name	Weight
0	setContainmentGen	0.0
0	setContainment	0.0
0	delete	0.0
1	setEOpposite	0.0
1	setUpperBound	0.0
1	delete	0.0

Table 3.3: The empty Q-table for error 3.

Context	Action name	Weight
0	delete	0.0
1	setName	273.77570694444444
1	delete	0.0
2	setName	392.24155
2	delete	0.0

Table 3.4: Q-table for error 1 in the fifth episode.

optimal. As a result, the chosen action depends on the implementation. This implementation chooses the first action in the table `setEType`. It receives a reward of 100 based on the preferences and calculates the new Q-value to the same value.

The next step attempts to solve Error 3. It chooses the optimal action, but all the values for this error is also zero. It takes the action `setContainmentGen` and receives a reward of 100. The new Q-value has a slightly different computation when there are no errors left to fix. It is not possible to calculate the maximum reward for the next step, because no more steps remain. The computation then looks like this:

$$Q_{\texttt{Finished}}(s_{\texttt{Error 3}}, a_{\texttt{setContainmentGen}}) = Q_t(s_t, a_t) + \alpha\,(r - Q_t(s_t, a_t)) \quad (3.4)$$

$$= 0 + 1 \cdot (100 - 0) \quad (3.5)$$

$$= 100 \quad (3.6)$$

The first episode received a total reward of 400. Next, the algorithm starts another episode. The Q-table is kept from the previous episode, but the model is reset with all the errors. The first episode is Error 1, and this time the algorithm chose the optimal action `setName` with a weight of 200. The new Q-value is calculated, but now the alpha has a new value. This is because the implementation used in this demonstration gradually decreases the alpha. In other words, the algorithm is less and less influenced by new discoveries.

$$Q_{t+1}(s_t, a_t) = 200 + 0.9608333333333333\,(200 + 1 \cdot 100 - 200) \quad (3.7)$$

$$= 296.0833333333333 \quad (3.8)$$

For the following steps in the episode the algorithm chose the optimal action, both being the same options as the first episode. The algorithm continues to run episodes, sometimes mixing it up with random actions. When the algorithm starts on Error 3 in the fifth episode, the Q-tables look like table 3.4, 3.5 and 3.6. In the third step the algorithm chooses a random action `setUpperBound`, which sets an unknown error as the next step. This new error states that the lower bound 5 must be less than or equal to the upper bound 1.

- **Error 4**: The lower bound 5 must be less than or equal to the upper bound 1

Context	Action name	Weight
0	setEType	189.82506752083333
0	delete	0.0

Table 3.5: Q-table for error 2 in the fifth episode.

Context	Action name	Weight
0	setContainmentGen	89.1325
0	setContainment	0.0
0	delete	0.0
1	setEOpposite	0.0
1	setUpperBound	0.0
1	delete	0.0

Table 3.6: Q-table for error 3 in the fifth episode.

The algorithm notices it does not have any entries in the Q-table for this error and initializes it with actions provided by EMF. The Q-table now contains an entry for the newly introduced error with the actions displayed in table 3.7. After the initialization is complete, it finishes the calculation of the new Q-value for `setUpperBound`.

Since the model still contains errors a fourth step is executed. It chooses the optimal action `delete` in context 0 and receives a reward of -1000. The Q-value for the action is computed and stored in the Q-table. The problem with this delete-action is that the opposite reference in the `Heater`-class is deleted. This does not fix the error, so in the next step the algorithm is still attempting to solve it. This time the optimal action is `setLowerBound`. However, after deleting the opposite the algorithm is unable to apply the action. It still receives a reward of 100 and calculates a Q-value of 84,33. This makes `setLowerBound` the optimal action, and the algorithm gets temporarily stuck in a loop as it cannot be applied to the model. This is ended by reaching the maximum number of steps or by a random action. In the execution, step 12 chose to delete in context 0 again which deleted the entire reference and the error disappeared.

In episode 10 the algorithm chooses the optimal action `setName` for error 1, the optimal action `setEType` in context 0 for error 2, the optimal action `setUpperBound` in context 1 for error 3 which again introduces error 4 to the model. This time the algorithm chooses the optimal action `setLowerBound` and is able to apply it, not having deleted the opposite. This gave the best reward of all the solutions with a value of 500 and resulted in a model like figure 3.4.

Context	Action name	Weight
0	delete	0.0
1	setLowerBound	0.0
1	delete	0.0

Table 3.7: The empty Q-table for error 4.

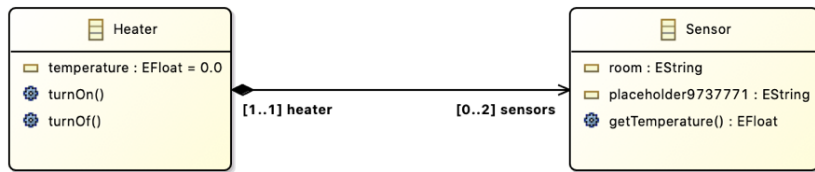

Figure 3.4: A model with the errors solved by the algorithm.

The model is not perfect, and some need for human completion is needed. For example, the duplicate Room-attribute is now named *placeholder9737771*, and the developer has to delete it or name it appropriately.

3.2 Code origin

This research project did not start from scratch. [8, 9, 10] proved that it is possible to repair models with Q-learning by building an algorithm and displaying its potential by repairing 100 broken models. That algorithm was the starting point of the repairing tool.

The original code was, unfortunately, difficult to build and expand upon due to its complexity and the lack of interface towards the algorithm. It was constructed with most of the code in one class containing 1612 lines of code. [37] states that classes should be kept small, because as systems become more complex, they will take more time for other developers to understand.

In the initial phase of the refactoring, different responsibilities were identified and separated into classes. This process proved effective in gradually building an understanding of the system.

Whilst this project started, the authors of [8, 9, 10] was still continuing their development of the algorithm. A tool for allowing both projects to continue the development without causing problems for each other, whilst still being able to retrieve the latest versions of the working code from each other was needed. Git, a distributed revision control system [55], proved very helpful in dealing with this challenge.

Git allows to *fork* a repository, creating a personal copy of the code base. This new copy is still connected to the original code base, and new code can be exchanged between the two versions. However, the future of the two projects and the commonality between them was a little ambiguous at the beginning of this project. As a result, we opted to make the common algorithm a separate Git project and put other artifacts like execution scripts and models not relevant to the algorithms function in an isolated project. This separation made it possible to include the algorithm as a *submodule* in other projects without getting unnecessary files and models. The common code can now be used and improved upon and different projects can be built on top of it. These projects do not break if the underlying algorithm changes, as Git holds submodules in *detached head state* meaning changes need to be actively fetched.

The distance algorithm described in section 3.10 was also added as a submodule. The reasoning behind this was twofold. Firstly, Git allows quick updating of the distance algorithm if any improvements are made to the distance project. This would require more manual work if the code had been copied into the Parmorel project. Secondly, Git now shows where the code comes from to other developers, and it can be tracked back to its original creators. Because the Parmorel tool is created for a thesis, this felt like an appropriate way to credit the original code authors in the repository.

Never having worked with submodules before, two minor irritations was experienced considering this way of structuring the code. Firstly, the code in the submodules are not fetched automatically with the `git clone` command. It leaves an empty folder, rendering a lot of the code with errors as references to classes in the submodules break. To clone everything, the user must either use a `recursive`-flag or pull each submodule manually. This might not be intuitive to users who checks out the project. Secondly, all changes to the code in a submodule is pushed back to that submodule. This might seem obvious but caused an issue when the distance project was altered slightly to comply with Parmorel. The original repository did not provide write permissions, so the required changes could not be pushed. As a result, each time the Parmorel code base was pulled into a new workspace, the required changes to the distance algorithm had to be made locally. This was resolved by forking out a copy of the code, resulting in a new repository with write access that could be updated as desired.

3.3 Development method

> One of the best ways to ruin a program is to make massive changes to its structure in the name of improvement [37].

As stated in section 3.2 this project did not start from scratch but expands upon the work done in [8, 9, 10]. To avoid breaking the algorithm, it was a desire to work in the discipline of Test-Driven Development (TDD). TDD is a programming practice where test cases are written before the production code. One of the advantages with TDD is that any regression errors introduced when modifying or adding features to the program is detected when running the tests [27]. This gives the programmer a confidence that no new errors have been introduced to the program. In this project, TDD could provide a verification that the behavior of the system remained unchanged while refactoring.

To work with TDD a test suite of automated tests that could verify that the behavior of the system remained unchanged was required [37]. This test suite should consist of tests at different levels. The unit tests make sure the individual software components does the right thing in isolation, the integration tests combine units to make sure they work correctly together, and acceptance testing makes sure the software works as the specifications required [27].

These testing levels are each different part of the V-model of software testing [39]. The V-model illustrates that while programming it is unit tests that are executed, and integration tests are executed after the unit is finished to make

sure it functions as expected. Unfortunately, it was almost impossible to create unit tests for the initial code because the majority of the code was in units too big to write unit-tests for. Instead, we opted for using the initial repairing script for fixing 100 models as an integration test. If this still executed without problems, we considered the behavior unchanged. This might not cover all code paths and is therefore a lot less accurate, but it made sure the behavior did not diverge massively from the original code.

3.4 Code structure

The project is created in the Eclipse IDE using the programming language Java, where the classes are organized into packages. The Parmorel algorithm contains 6 different packages illustrated in figure 3.5 and is available on GitHub from the link specified in appendix A.

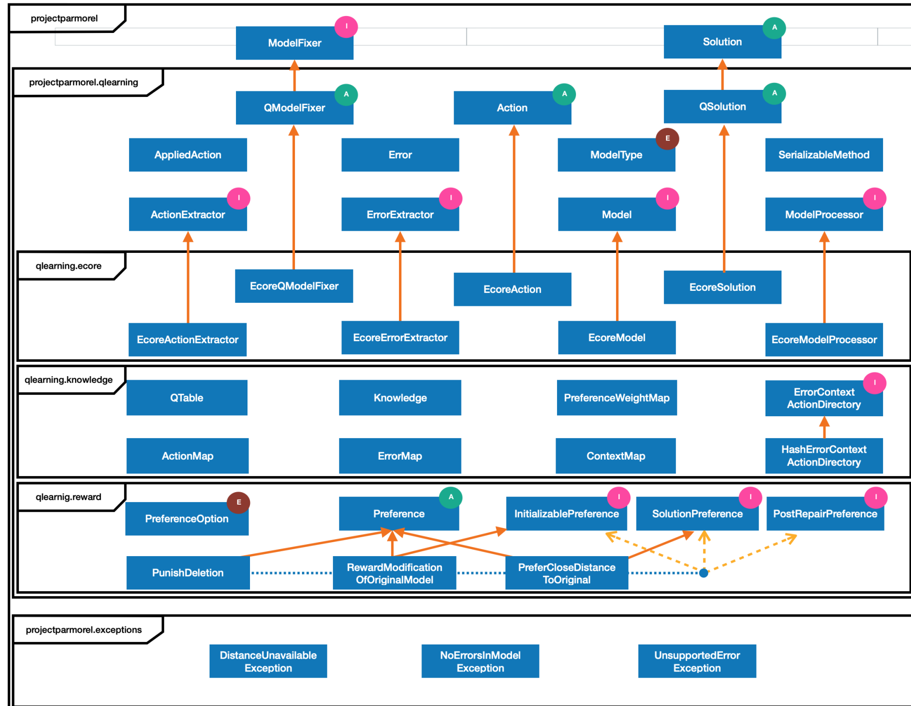

Figure 3.5: The package structure of the algorithm annotated with *I* for interfaces, *A* for abstract classes and *E* for enums. The arrows show inheritance. Note that not all preferences are included in this illustration.

The package `no.hvl.projectparmorel` is the top package and contains all the other packages. In this package, the interface *ModelFixer* and the abstract *Solution*-class returned from methods in this interface are contained. These artifacts are meant to be general so that they can be implemented by other model repair approaches, and still use the same execution scripts (section 3.9).

Plug-ins and other artifacts depending on these classes should not have to change a lot if the underlying algorithm is changed.

All the custom exceptions that can be thrown in the Parmorel algorithm are located in the `hvl.projectparmorel.exceptions` package.

`no.hvl.projectparmorel.qlearning` is a package containing the Q-learning related classes. This includes the `QModelFixer` that implements the *ModelFixer* interface. Tree more packages are organized inside the qlearning package. Two of them are `no.hvl.projectparmorel.qlearning.knowledge` and `no.hvl.projectparmorel.qlearning.reward`. Inside `knowledge`, all the classes regarding the knowledge (section 3.8) are stored. The classes that handles rewards are located in the `reward`-package. This includes preferences, as they are the source of which rewards are derived. The reason that `knowledge` and `reward` are located inside the `qlearning` package is that only the Q-learning algorithm should be dependent on their contents. Other model repair algorithms will probably structure the knowledge and reward in other formats.

The last package is `no.hvl.projectparmorel.qlearning.ecore`. Inside this package, all the abstract classes and interfaces dependent on the metamodel are implemented with respect to the Ecore metamodel. To ensure that the rest of the algorithm is metamodel independent, the files in this package are the only ones that are allowed to import from emf-packages.

3.5 Actions

An action is some operation being applied to the model. Every action has a unique id, a name, a method, a context ID and a weight. The method is the action itself and contains a `java.lang.reflect.Method` that can be called by the algorithm to change the model. The context ID is what context the action is applied to. This corresponds to the contexts in the error, and one action instance can only correspond to one context. This is because an action is considered for each context, and not all methods are suitable for all of them. Say two elements have the same name, this can be solved by setting a new name or deleting one of them in the context of the attributes. In the context of the containing class, a rename would not solve the error. However, deleting the class will. Additionally, the method might have different weights for different contexts.

When actions are applied to the models, they often need some parameters. `setEType`, `setName` and `setUpperBound` are some of the actions chosen by the algorithm in section 3.1. How does the algorithm select what type to give an attribute, or what name to give a duplicate? Our implementation is not advanced in this area and has a set of standard parameters to use in the methods.

If the action requires parameters, the method shown in listing 3.1 is called. It receives a list of all the parameter types and returns a list of default values corresponding to the parameter type. All attributes with a missing type will be set to `EString`, all bounds will be set to 1 etc.

```
660    private Object[] getDefaultValues(List<String> list) {
```

```java
661     List<Object> values = new ArrayList<Object>();
662     Random rand = new Random();
663     for (int i = 0; i < list.size(); i++) {
664       if (list.get(i).contentEquals("int")) {
665         values.add(1);
666       }
667       if (list.get(i).contentEquals("boolean")) {
668         values.add(false);
669       }
670       if (list.get(i).contentEquals("booleanTRUE")) {
671         values.add(true);
672       }
673       if (list.get(i).contains("String")) {
674         values.add("placeholder" + rand.nextInt((999999 - 1) + 1) +
              1);
675       }
676       if (list.get(i).contentEquals("org.eclipse.emf.ecore.
            EClassifier")) {
677         values.add(EcorePackage.Literals.ESTRING);
678       }
679       if (list.get(i).contentEquals("org.eclipse.emf.ecore.
            EClassifierCLASS")) {
680         values.add(EcorePackage.Literals.ECLASS);
681       }
682       if (list.get(i).contentEquals("org.eclipse.emf.common.notify.
            NotificationChain")) {
683         values.add(new NotificationChainImpl());
684       }
685       if (list.get(i).contains("TypeParameter")) {
686         values.add(EcoreFactory.eINSTANCE.createETypeParameter());
687       }
688       if (list.get(i).contains("Reference")) {
689         values.add(EcorePackage.Literals.
              EREFERENCE_EREFERENCE_TYPE);
690       }
691       if (list.get(i).contains("Literal")) {
692         values.add(null);
693       }
694     }
695     Object[] val = new Object[values.size()];
696     val = values.toArray(val);
697 
698     return val;
699   }
```

Listing 3.1: The method providing default parameters.

This is sufficient to make the algorithm work, but it would be interesting to see if ML could be applied to this part of the algorithm as well. This is left for further work.

3.6 Rewards

The rewards given to the algorithm can have different weights. A weight of 200 is a greater reward than a weight of 100. If a reward has a negative weight this is considered a punishment. The preferences selected by the user is represented by weights in the algorithm. Currently, these weights are selected by trial and

error figuring out what worked well and what did not. This can be changed in the implementation to allow more user control, discussed in chapter 9.

When the user selects a solution, the plug-in will give a reward to all the actions used to construct it. If a solution has used an action several times, the action will be rewarded correspondingly. By allowing this post repair interaction, the user is allowed to influence how the algorithm learns without relying heavily on strong user interaction during the repair process. Furthermore, the user has more information available after the repair process is complete and can compare the different solutions and reward the best one.

3.7 Errors

As explained at the beginning of this chapter, the Q-table is populated with actions for an error when it is encountered by the algorithm for the first time. To make sure the error is supported, the algorithm checks if the error still exists after an action is applied. If the error is gone, the action was successful and is added to the Q-table. If none of the provided actions removed the error, it is marked as unsupported. Detecting if an error still exists is done by comparing a list of all the errors residing in the model before and after an action is applied but this is not a trivial task.

An error consists of a numeric error code unique to a specific type of error, a message explaining the error in a human readable form and a list of contexts describing the error. An example is an error with code `32` and the message `There may not be two features named 'address'`. The lists of context would contain the object that is the primary source of the problem (i.e. the attribute named address), an object describing the problematic feature or aspect of the primary source (i.e. the attribute with the same name), and the remaining elements are additional objects associated with or describing the problem (e.g. the class containing the attributes).

Problems may occur if all the components of an error are compared to see if their values are the same. Some of the messages describing the error contain information dependent on a memory address causing the message to be different for the exact same error every time the error is extracted from the model. As a result, the errors will not be equal, and the algorithm will not discover that the error still exists in the model. This leads to the algorithm thinking the action worked for the error, whilst in reality it did not.

If only the error code is used to compare errors, problems arise when models have several of the same error type. Let's say we have a model with several errors with code 44, indicating that names are not well formed. Even if the algorithm fixes one of these errors, the error code 44 will still remain in the model as there still exists names that are not well formed. This results in the algorithm thinking the action did not work for the error, even if it really did.

The current implementation solves this by counting the number of times the error code occurs in the model before and after an action is applied. If the error being fixed has less occurrences after the action is applied, the algorithm can

assume the action worked.

Unsupported errors will be ignored at repair time by the algorithm. The unsupported errors for EMF are usually related to proxies (i.e. elements existing in other resources) and namespaces. Although EMF does not provide actions to solve all errors, the possibility of adding custom actions are discussed in chapter 9.

3.8 Representation of knowledge

In the original code, the knowledge gained by the algorithm was stored in a triple `HashMap`. In the first map, the key was the error code. This would return a second map, were the key was the context that again returned a third map where the action id was used as a key to get the action.

This data structure worked in the sense that it was fast and kept the actions for different error codes or contexts separate from each other. The problem was that it was very hard for programmers unfamiliar with the project to understand how to use it. Listing 3.2 shows how actions were added to the Q-table.

```
155    if (!getNewXp().getqTable().containsKey(e.getCode())) {
156      d.put(a.getCode(), weight);
157      dx.put(num, d);
158      getNewXp().getqTable().put(e.getCode(), dx);
159      if (!getNewXp().getActionsDictionary().containsKey(e.getCode
           ())) {
160        hashaux.put(a.getCode(), new ActionExp(a, new HashMap<
             Integer, Integer>()));
161        hashcontainer.put(num, hashaux);
162        getNewXp().getActionsDictionary().put(e.getCode(),
             hashcontainer);
163      } else {
164        if (!getNewXp().getActionsDictionary().get(e.getCode()).
             containsKey(num)) {
165          hashaux.put(a.getCode(), new ActionExp(a, new HashMap<
               Integer, Integer>()));
166          getNewXp().getActionsDictionary().get(e.getCode()).put(
               num, hashaux);
167        }
168
169        else if (!getNewXp().getActionsDictionary().get(e.getCode()
             ).get(num).containsKey(a.getCode())) {
170          getNewXp().getActionsDictionary().get(e.getCode()).get(
               num).put(a.getCode(),
171               new ActionExp(a, new HashMap<Integer, Integer>()));
172        }
173      }
174    }
```

Listing 3.2: Parts of the original code to add an action to the Q-table if it is the first time an error is encountered.

Listing 3.2 only covers the scenario that the error code is encountered for the first time. If the Q-table already contained the error code only the code from line 164 would be necessary and were implemented again in a different `if`-branch. This illustrates one of the problems with the old `Map`-structure. The

programmer always had to check if there existed a `Map` for the key. If it did not, a new `HashMap` had to be created. This resulted in many lines of code that was unnecessarily hard for programmers to understand, given the need to keep track of what values should be placed where.

When refactoring, alternative structures was considered. Primarily two structures were discussed: the existing `Map`-structure and a tree-based structure.

3.8.1 Tree structure

A tree-structure is a hierarchical data structure constructed by nodes [35]. A node can have an arbitrary number of sub-nodes, referred to as children. If the knowledge were to be implemented as a tree, it might look like figure 3.6. In this implementation, the first layer of nodes represents the error codes. Each error can have several child nodes that each represents a context where the error resides. Each of these context nodes can again have several children nodes, each representing an action that might fix the error.

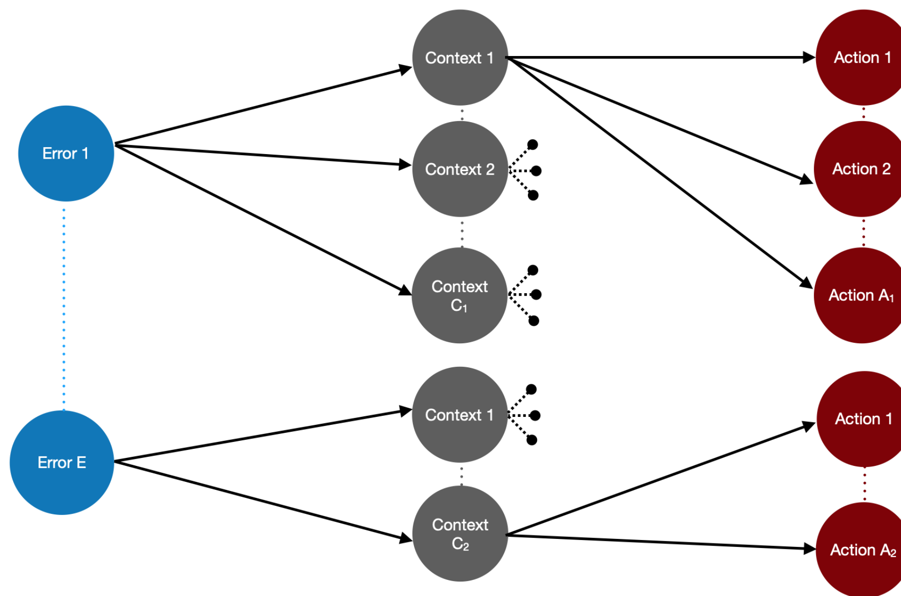

Figure 3.6: Visualization of a tree structure.

The advantages found with this approach was how easily it could be visualized, making it understandable for new developers. In reality, however, this implementation does not differ much from the `Map`-structure. The only difference between the implementations is that the information is stored as a node in a tree, rather than an entry in the `Map`. The programmer would still have to navigate between nodes to access the information.

3.8.2 Map structure

We chose to implement the knowledge in a `Map` containing other maps like the original code. This can be visualized similarly to the tree structure, although it looks slightly more complex as shown in figure 3.7. The main difference between the structures is that we do not have to iterate over the child nodes to find the one we want when traversing a tree. Instead we can look up what value we want from the `Map`, which results in a slightly faster implementation.

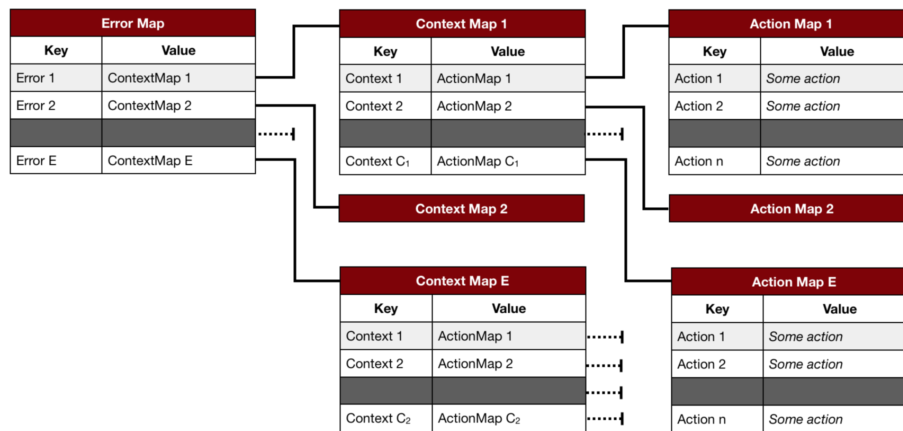

Figure 3.7: Visualization of a `HashMap` structure.

The main difference from the original implementation was an interface added to the structure, providing more intuitive method names. This makes it easier for other programmers to understand the system, as they don't have to depend on or even understand the implementation. This is illustrated by listing 3.3, where a caller only references the directory with the `addAction`-method. Interfaces also makes it easy to change the implementation later, for example to a tree structure, without breaking the code depending on it [37].

By wrapping the inner `Map`-structures in classes, a lot of the logic could be more expressive rendering the detailed implementation easier to understand. An example of this is the class `ErrorMap` in listing 3.4. The `ContextMap`-class behaves very similar to the `ErrorMap`-class, but handles the next layer in the data structure, mapping from context id's to possible actions.

```
10 public class HashErrorContextActionDirectory implements
       ErrorContextActionDirectory {
11   private ErrorMap errors;
12
13   public HashErrorContextActionDirectory() {
14     errors = new ErrorMap();
15   }
16
17   @Override
18   public void addAction(Integer errorCode, Integer contextId,
         Action action) {
19     errors.addAction(errorCode, contextId, action);
20   }
75 }
```

Listing 3.3: Base class of the knowledge structure.

```
15  class ErrorMap {
21    private Map<Integer, ContextMap> contexts;
22
23    protected ErrorMap() {
24      contexts = new HashMap<>();
25    }
116   protected void addAction(Integer errorCode, Integer contextId,
        Action action) {
117     if (contexts.containsKey(errorCode)) {
118       contexts.get(errorCode).addAction(contextId, action);
119     } else {
120       contexts.put(errorCode, new ContextMap(contextId, action));
121     }
122   }
184 }
```

Listing 3.4: Implementation of nested `HashMap` wrapped in a class.

3.9 Extensibility

The algorithm is built and tested using the EMF API, but by implementing various interfaces and abstract classes the algorithm can be introduced to other model types. The Template Method pattern allows subclasses to redefine certain steps of the algorithm without changing the algorithms structure [25], and this is the basis for implementing new types of models. In this section the various required extension points will be explained, and figure 3.8 illustrates the various components in relation to each other.

By implementing the interface `ModelFixer` the entire Q-learning algorithm can be replaced with another type of algorithm. As explained in section 2.2.1 there are several ways to repair a model. If other solutions implement this interface, the plug-in will still work, and the test scripts and models used in Parmorel can be tested and compared to other repair approaches. The main methods in the interface is `fixModel(File model)` which returns a `Solution`. If the repair approach produces several solutions, these can be obtained through the method `getPossibleSolutions()`. If only one solution is produced, the list should only contain that solution.

The `Solution` is an abstract class describing the final result of the repair process. It contains a file referring to the original model and a file referring to the repaired version. This is not dependent on the metamodel, but the `calculateDistanceFromOriginal()`-method on the solution object is. This method calculates the editing distance between a suggested repaired version and its original. The distance is currently only used by some specific preferences in the Q-learning algorithm, and if there is no intention of using these preferences or provide the user with the distance information a `DistanceUnavailableException` can be thrown.

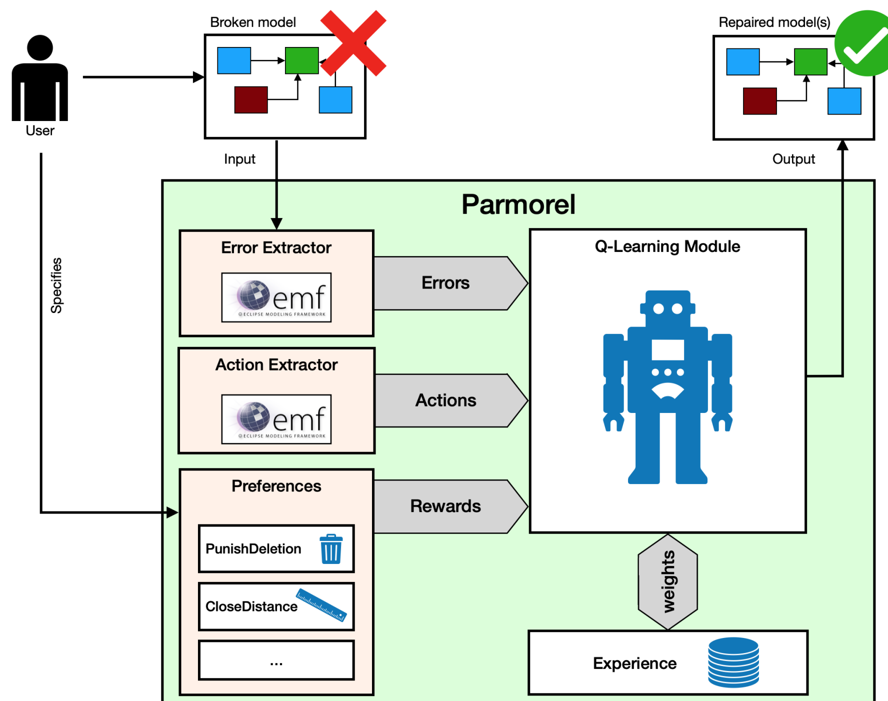

Figure 3.8: Parmorel components.

This thesis focuses on Q-learning and has implemented the `ModelFixer` interface in an abstract class called `QModelFixer`. This class is abstract in order to handle all the logic regarding Q-learning but separate the algorithm from the Ecore metamodel. This is one of the main classes that needs implementation in order to support other metamodels. The derived classes do not contain much logic themselves but are responsible for providing the `QModelFixer` with metamodel specific components.

One of these components is a `Model`. This wraps the model representation, allowing the `QModelFixer` to handle the model whilst delegating the metamodel specific implementation to specific components. Two of the methods are `getRepresentation()` and `getRepresentationCopy()` which both returns an `Object`. Changes can be made to the copy without altering the original model, but any changes made to the representation might. `Object` is the root of the Java class hierarchy, meaning all classes has `Object` as a superclass [45]. This allows the model to be represented by any Java class, and the only important aspect is to return the same type of class as the other metamodel specific components expects as input. The Ecore-implementation, for instance, returns an `org.eclipse.emf.ecore.resource.Resource`-object. The last two methods on a `Model` are `save()`, which saves the representation, and `getModelType()`. `ModelType` is an `Enum` specifying what type of model it is. Different metamodels might have different problems that cannot be handled, so each Enum type contains a set of unsupported error codes. If the developer knows of any problems, these can be added to the Enum type constructor when it is implemented.

Otherwise, the Parmorel algorithm will automatically add error codes to this list if no actions from the modeling framework resolves the error.

ActionExtractor and ErrorExtractor are two components specific to the metamodel. The ActionExtractor provides Parmorel with all the actions that can be applied to the model in order to fix a list of errors. In order to speed up the algorithm, methods that do not result in any change (get-methods etc.) should be omitted. The Action class is also abstract, but isDelete() and getActionType() are the only methods that depends on the metamodel. ErrorExtractor simply returns a list of the errors in the model.

The last component specific to the metamodel is the ModelProcessor. This is the class responsible for applying actions to the model. There is a lot more behind this task than meets the eye. Applying actions include determining if the method requires parameters and what they should be.

In the process of initializing the Q-table with new actions, the actions are applied to the model to see if they remove the error from the model. If the action does not get rid of the error, it is not added to the Q-table in order to reduce the search space for the algorithm when looking for an action during the repair process. Because the Q-table initialization is closely related to applying actions to the model, this task is also delegated to the ModelProcessor.

After all the components above are implemented for a metamodel, the algorithm should be able to handle all models derived from that metamodel.

The preferences used by Parmorel can be altered or supplied with completely new preferences by extending the abstract class Preference and adding an enum constant to PreferenceOption[1]. The Preference contains a method called rewardActionForError(Model model, Error error, Action action) which is called for every action the algorithm applies to the model, and returns the reward given from the preference in the form of an integer. Not all the parameters are required in generating the reward, depending on the preference. Some preferences require additional method calls for initialization, post processing etc., and can be achieved by implementing various interfaces. An overview of the algorithm and at what point the different methods regarding preferences are called can be viewed in figure 3.9. Lastly, the PreferenceOption must be added to the switch-statement in the RewardCalculators initializeFrom(List<PreferenceOption> preferences)-method with the corresponding preference implementation.

Some preferences might want to compare the model being repaired with the original or with the state of the model just before the action is applied. If this is the case, the preference should have some initializing call to store information about the model prior to any changes made to it. In order to accomplish this the preference must implement the interface InitializablePreference. This has two method calls, both with the model as a parameter and allows the preference to store the information required in order to calculate the reward at a later stage. The first initializing call is made prior to any episodes having

[1] A video demonstration is available here: https://github.com/MagMar94/ParmorelExperimentResults/wiki/Creating-custom-preferences

started (Initialize preferences[1] in fig. 3.9), and the other is called just before selecting an action to apply to the model (Initialize preferences[2] in fig. 3.9). It is usually necessary to write model-specific code in these preferences, as the model only returns an `Object`. In order to maintain extensibility, it is recommended to throw an exception that explains what has happened if someone were to use the preference for an unsupported metamodel. An example of this can be seen in listing 3.5, where the preference depends on the initial number of errors. If the model conforms to the Ecore metamodel the preference works as expected, but if it does not an exception is thrown.

```
9  class PunishModificationOfModelPreference extends Preference
       implements InitializablePreference {
10
11   private int numbersOfErrorsBeforeApplyingAction;
12   private ErrorExtractor errorExtractor;
13
14   public PunishModificationOfModelPreference(int weight) {
15     super(weight, PreferenceOption.PUNISH_MODIFICATION_OF_MODEL);
16
17   }
18
19   @Override
20   public void initializePreference(Model model) {
21     switch (model.getModelType()) {
22     case ECORE:
23       errorExtractor = new EcoreErrorExtractor();
24       break;
25     default:
26       throw new UnsupportedOperationException("This preference is
             not yet implemented for this model type.");
27     }
28
29   }
30
31   @Override
32   public void initializeBeforeApplyingAction(Model model) {
33     numbersOfErrorsBeforeApplyingAction = errorExtractor.
           extractErrorsFrom(model.getRepresentationCopy(), false)
34         .size();
35   }
```

Listing 3.5: An example of how a preference can maintain extensibility to support other models.

Some preferences do not look at individual steps, but rather at the finished solution. If this is the case, the preference should implement the interface `SolutionPreference`. The method in this interface is called once a complete solution is derived, with the complete solution as a parameter. Additional parameters include the model (in the same way as in `InitializablePreference`) and the Q-table. The reason the preference gets the Q-table is that all the actions used to create the solution have received a weight whilst repairing the model, and these weights can now be altered in retrospect considering the final result. To make sure the Q-table is updated correctly with respect to the existing reward it is recommended to leave to the altering of the table to the `rewardAction` method available through the abstract `Preference` class.

The last type of preference available is one that compares all the finished solu-

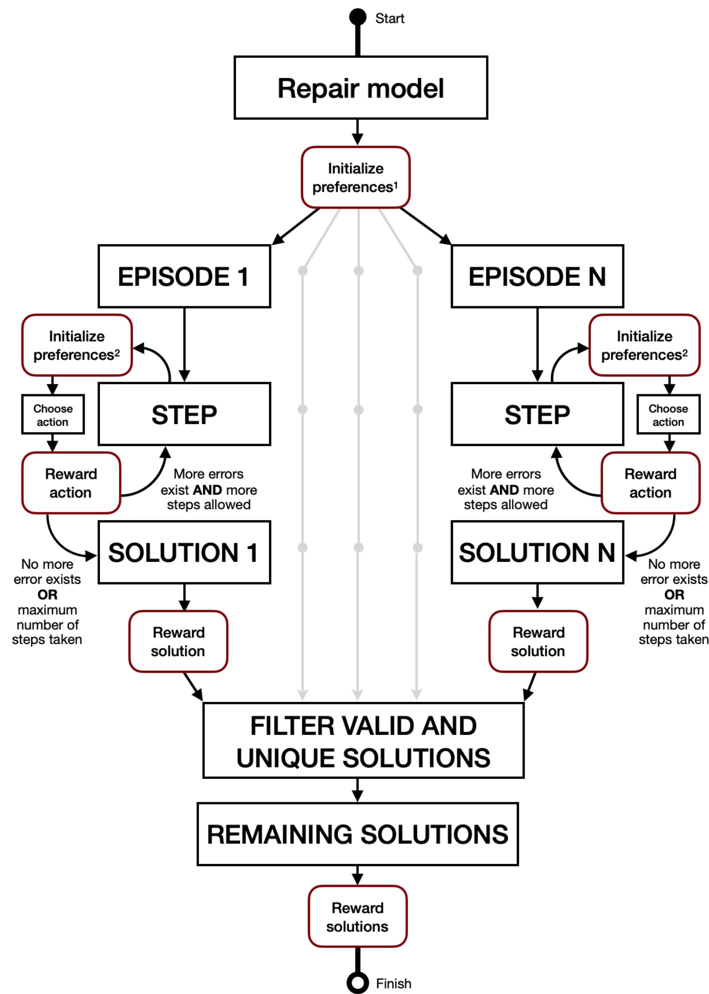

Figure 3.9: An overview of where in the repairing process the different preference-methods are called.

tions. Preferences that requires this functionality implements `PostRepairPreference`, which method is called with a list containing all the solutions as a parameter, along with the Q-table for post-repair updates. This is useful for preferences like prefer long or short sequences of actions because long and short are circumstantial. Whether the action sequence is long or short can easily be deduced post-repair by comparing the length of all the action sequences. This differs from `SolutionPreference` because it is calculated after the repair has finished producing solutions and is thus not able to use the knowledge in the current repair.

When the algorithm is executed the user might select to combine preferences. This is handled by the `RewardCalculator` by looping over all the selected preferences and adding up the total.

3.10 Edit distance

As mentioned in section 2.1.5 it is possible to measure the distance between two models. A distance measure has been implemented in the plug-in to see if knowing the distance between a broken model and the repair alternatives could be beneficial for the developer when selecting the optimal solution. The distance algorithm is based on EMF Compare [19], with a custom matching engine provided by [1]. The matching engine is responsible for reporting what elements are equal to each other, and the distance is calculated based those matched elements.

Once the distance calculation was implemented, the concept was taken a step further by implementing a preference working to minimize the distance to the original model. In order to minimize undesired side-effects when repairing models, [33, 59] has highlighted the importance of preserving the model structure. Knowing the distance between the proposed solution and the original model can be a powerful tool in order to accomplish this [1, 32, 58]. Parmorel uses the distance value as a reward, being a high value for similar models, resulting in the framework learning how to repair models with the result being as close to the original as possible.

EMF Compare is an Eclipse project that provides support for model comparison and merging of EMF models, in a generic and customizable way [19]. The comparison process is split into different activities, and these activities are handled separately by different software entities, i.e. *engines*. EMF Compare provides full support to create custom or metamodel-specific solutions by extending its default behavior, resulting in a completely customizable comparison process.

The matching engine provided by [1] calculates the similarity between two given model elements by integrating ontological information, i.e. information showing the relations between the meaning of the words in English. This ontological information is extracted from the WordNet lexical dictionary [47]. Words are grouped into sets of synonyms called *synsets*, including a generic definition tying the words together. The set also includes information specifying semantic relationships connecting the contained words to other synsets. Some of these relationships constitute *is-a-kind-of* and *is-a-part-of* hierarchies. For example, a beer ontology can be used to determine that Hansa is a kind of Pilsner which is a kind of beer. Hops is a part of beer. These semantic relationships apply to all members of a synset, as their meaning is the same. Words can also be connected to other words through lexical relations, e.g. antonyms. By using this ontological information, the engine can compare model element names and calculate a semantic distance.

Distance values ranges from 0 to 100, with 100 meaning the models are identical with respect to the properties included in the calculation. A model with multiple repair solutions can have different distances for the various solutions. An example from the ACMR set (introduced in chapter 5) has identical errors in two classes, displayed in figure 3.10. Parmorel finds four possible actions for each error:

- Set the upper bound to 1

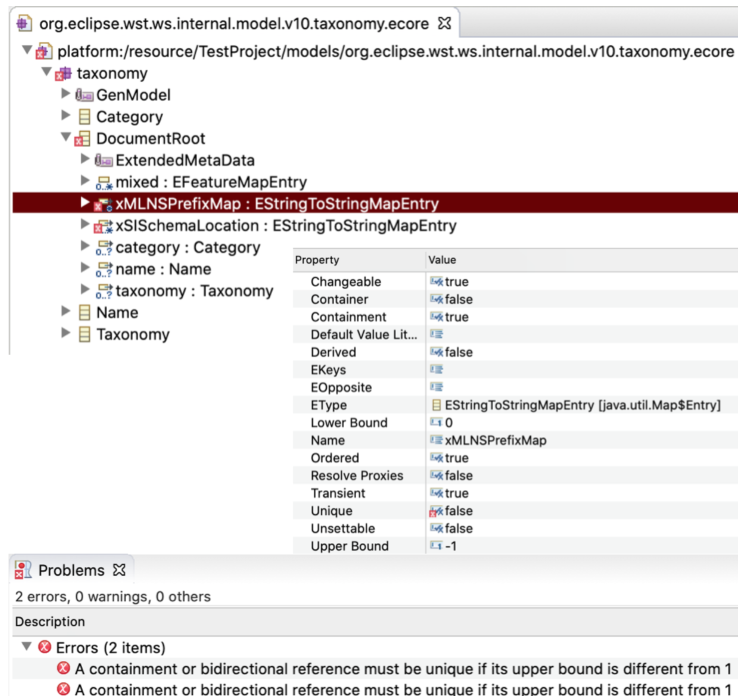

Figure 3.10: An overview of the model org.eclipse.wst.ws.internal.model.v10.taxonomy.ecore and its residing errors.

- Set the unique property
- Delete the faulty reference
- Unset the containment property

By combining these actions for the two errors a total of 16 solutions can be obtained. These solutions will result in different distance values, and a subset of these are highlighted in table 3.8. Solution 1 and 4 are closest to the original because they alter the uniqueness and the containment properties, and neither of these are considered by the comparison matching in question. However, solution 2 and 3 modify properties considered in the matching comparison: a modified upper bound and a deleted reference. This will have an impact on what actions gets chosen by Parmorel, and as a result the user is provided with an additional way to impact the repair procedure. By changing or adapting the distance algorithm, making it take into account more properties or be less strict, Parmorel will adapt the chosen actions accordingly.

Another distance algorithm tested in this thesis was the `EditionDistance`-class implemented in `EMF Compare` [23]. However, this implementation did not work for Parmorel as it always returned a distance of zero even though there where changes to the model. When the Eclipse Community Forum was consulted, the reply stated that the required use case was not what they had in mind when implementing it [22].

Solution number	**Actions**	**Distance**
1	a_1: unset containment a_2: unset containment	100
2	a_1: change upper bound a_2: change upper bound	89
3	a_1: change upper bound a_2: delete reference	89
4	a_1: set unique a_2: set unique	100

Table 3.8: The distance resulting from actions applied to the model displayed in fig. 3.10.

Chapter 4

Use cases

In this chapter, some common use cases and corresponding workflow for the plug-in are presented.

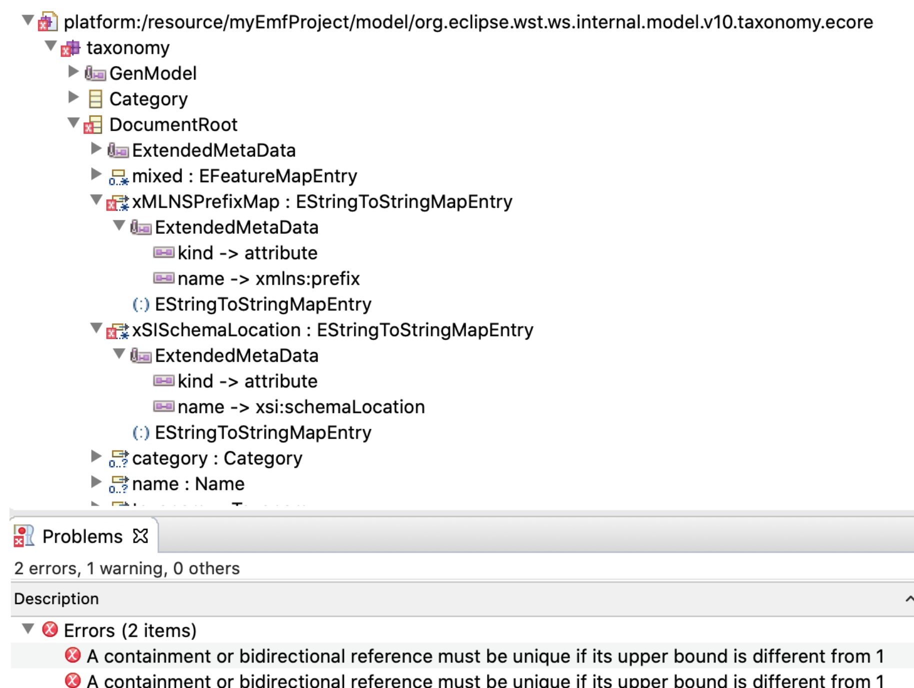

Figure 4.1: A model containing two errors.

The first use case is when an error occurs in a model being worked on by a developer, as the example illustrated in fig. 4.1[1]. In order to get suggested solutions on how to repair the error, the developer triggers the repair action in the Parmorel plug-in and selects some preferences (an example of the preference

[1] A video demonstration is available here: `https://github.com/MagMar94/ParmorelExperimentResults/wiki/Using-Parmorel`

selection is illustrated in fig. 4.2). Different solutions are presented to the developer (fig. 4.3), and the preferred option is selected as a repair resulting in a consistent model that can be developed further. In order to understand the various solution alternatives, the user can view what actions have been taken in each of them (fig. 4.4), and the proposed repaired models can be compared to the original (fig. 4.5). The comparison view is provided by EMF and can be used to partially merge the two versions. If the user likes the solution, it can be selected as a repair replacing the original model and rewarding the chosen solution.

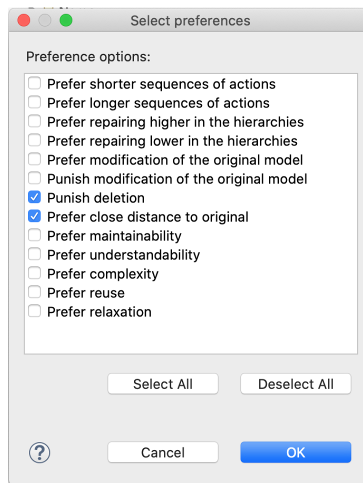

Figure 4.2: An example of a preference selection.

Another use case presents itself when a model contains several errors. They might have been introduced by a developer ignoring the error messages, postponing the repair to a later time, or by merging the work of several team members. Either way, someone has to handle the errors. The designated developer calls on Parmorel to present repair alternatives and solves all the errors in the model simultaneously.

A third use case is a developer not satisfied with Parmorel. The preferences do not allow the desired control over the repair process, and the developer concludes that additional preferences are required. These are created and added to the algorithm by the developer as explained in section 3.9[2], resulting in better

[2]A video demonstration is available here: https://github.com/MagMar94/ParmorelExperimentResults/wiki/Creating-custom-preferences

Select solution

Compare Show steps

#	Weight	Number of actions applied	Distance
1	398.0	2	100.0
2	386.0	2	94.0
3	-708.0	2	97.0

Compare solution Repair with selected solution

Figure 4.3: A list of potential solutions.

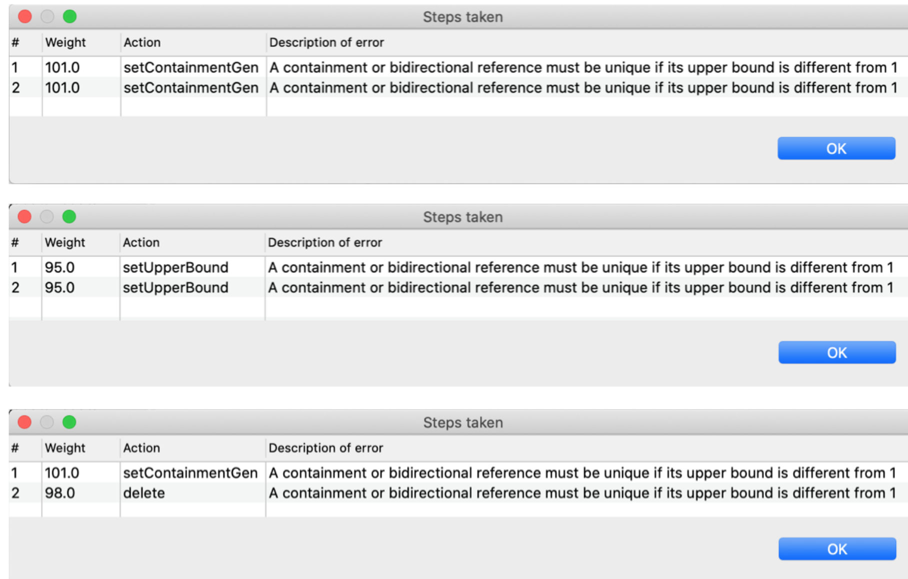

Figure 4.4: Three screenshots of the actions residing in each of the possible solutions present in the example.

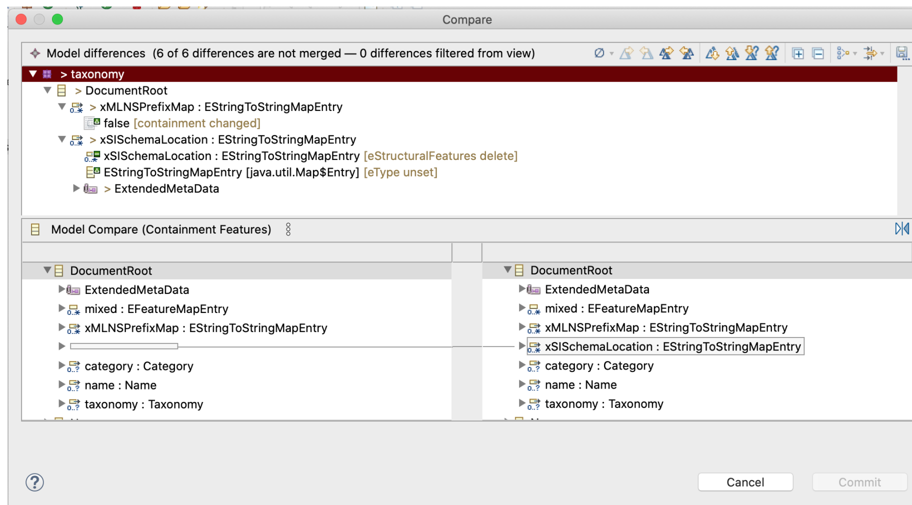

Figure 4.5: A screenshot comparing one of the proposed solutions (left) with the original (right).

control over the outcome for all future repairs.

If a project is based on another framework than EMF and this framework does not provide any tools for repairing the broken model, the Parmorel algorithm can be extended to handle those models as well. In this use case, the developer creates the required extensions explained in section 3.9 adopting Parmorel to handle the model type, whilst still using the existing Q-learning algorithm.

Another option is a developer not satisfied with Parmorel's Q-learning algorithm. However, the work process with the plug-in has become familiar and no other tool presents the desired algorithm. The developer decides to write a new algorithm, extending the `ModelFixer` interface presented in section 3.4 and making the necessary alterations to make the plug-in run with the new source code.

The last use case is another research proposing alternative solutions to repair model and want to compare its approach to Parmorel. By extending the same `ModelFixer` interface the same test-scripts can be run on the new solution, and the resulting models can be compared.

Chapter 5

Analysis and Assessment

In this chapter, the algorithm and plug-in developed in the thesis is assessed in experiments[1]. The experiments are divided into four activities described by [31]:

1. Definition of the objectives of the experimentation
2. Design of the experiments
3. Execution of the experiments
4. Analysis of the results/data collected from the experiment

When defining the objectives, a hypothesis is created describing what variables will be examined, and what metrics will be used to assess the results. The design phase involves planning the experiments, determining what conditions to apply (e.g. number of executions). Then the experiments are executed, and the results analyzed.

Two datasets containing broken models have been used to test the approach. The first set, named the AMOR set in this thesis, consists of 100 broken models achieved from breaking five consistent models of different sizes from GitHub with the AMOR Ecore Mutator 20 times per model [4]. This provides the models with some validity, as they are developed for a real-life purpose. As a result, they have the complexity and diversity of the models the Parmorel-tool can face in the industry. The disadvantage with this dataset is that the errors have not emerged naturally but have instead been introduced by an algorithm. This brings into question the authenticity of the errors, and whether or not they represent the errors the algorithm will face when utilized by developers.

The second set comes from the authors of [44], having searched through GitHub looking for Ecore-models and created a dataset containing 2410 models. From this dataset all the valid ones were detected and removed using the same `Diagnostician`-tool used for detecting errors in Ecore-models with Parmorel. This left 1184 models containing 31 different types of errors. Not all of these errors are supported by Parmorel because EMF does not provide actions to handle them. The

[1] The results of the experiments and the datasets can be found on GitHub: `https://github.com/MagMar94/ParmorelExperimentResults`

dataset was filtered again with the criteria that all the errors in the model could be handled by Parmorel. Unfortunately, this left only 6 models in the datasets. The reason for this turned out to be that many of the models contained error 4 that deals with unresolved proxies (i.e. elements existing in other resources), which is not supported by Parmorel. However, most of the models also had errors that Parmorel could handle. Finally, the dataset was filtered to only include models supported by Parmorel plus error 4. This left 107 models containing 12 different error types. This dataset is called the ACMR-set in this thesis (from its source **A**utomated **C**lassification of **M**etamodel **R**epositories [44]), and its supported errors with examples and explanations are listed in appendix B.

By repairing the models from these datasets, the algorithm can prove that it can handle errors introduced to models whilst they are being developed, and that it can handle large industry scale models.

5.1 Does preferences affect final model quality?

The algorithm is heavily dependent on the preferences, as they are the source of the rewards given for the chosen actions. RQ1 revolves around how much the preferences affect the proposed solutions. A hypothesis was created, stating that different preferences significantly affect the quality of the proposed solutions.

To calculate quality characteristics on models, some common metrics for the results were required. [12] provides a generic approach to assess the quality of models quantitatively, and proposes the following definition of maintainability:

$$Maintainability = \frac{NC + NA + NR + DIT_{Max} + HAgg_{Max}}{5} \tag{5.1}$$

The acronyms are explained in table 5.1. If the maintainability is easy the resulting numeric value will be low, whilst models with more difficult maintainability will result in higher values.

The understandability is adopted from [53] and complexity is adopted from [52], and defined as follows:

$$Understandability = \frac{\sum_{i=1}^{NC} PRED(C_i) + 1}{NC} \tag{5.2}$$

$$Complexity = NR - NUR + NOPR - Understandability + (NR - NCR) \tag{5.3}$$

$PRED(C_i)$ is the number of predecessor classes of the ith class. This matter because in order to understand a class one also has to understand its predecessors affecting it through inheritance. Both values are better if they are low, as this highlights better understandability and less complex models.

There are several ways to calculate reusability, and one of them is the attribute inheritance factor (AIF) proposed by [6]. This will give a high value for a high level of reuse, and can be defined as presented in [30]:

$$Reusability = AIF = \frac{INHF}{TNF} \tag{5.4}$$

Acronym	Meaning
AIF	**A**ttribute **I**nheritance **F**actor
DIT	the longest path from a class to the root of the containing generalization hierarchy
DIT_{Max}	the maximum DIT value obtained from each class in the model
$HAgg$	the longest path from a class to other classes in the relation chain
$HAgg_{Max}$	the maximum $HAgg$ value obtained from each class in the metamodel
$INHF$	the sum of all the **Inh**erited **F**eatures in the classes
NA	**N**umber of **A**ttributes
NC	**N**umber of **C**lasses
NCR	Number of TotalReference containment
$NOPR$	**N**umber of **Op**posite **R**eferences
NR	**N**umber of **R**eferences
NUR	**N**umber of **U**nidirectional **R**eferences
$PRED(C_i)$	the number of **pred**ecessor classes of the ith class
$REFint$	the difference between the upper bound and lower bound in a reference
TNF	**T**otal **N**umber of **F**eatures
UPB_{max}	the maximal **Up**per **B**ound of a set of references
UPB_{min}	the minimal **Up**per **B**ound of a set of references

Table 5.1: Explenation of the acronyms used in the quality characteristic formulas.

$INHF$ is the sum of all the inherited features in the classes, and TNF is the total number of features. It is preferable with a high reusability value.

The last quality characteristic used to measure the results are inspired by the metamodel relaxation concept [5], called *relaxation index* defined as follows:

$$Relaxation\ Index = \frac{\sum_{k=1}^{NR} REFint - UPB_{min}}{UPB_{max} - UPB_{min}} \tag{5.5}$$

The relaxation index indicates how strict a model is with respect to its cardinality constraints. A reference with its cardinality constraints set to [0..*] is more relaxed than $[i..i]$ (for $i \in \mathbb{N}$), because the developer has more freedom to define the number of instances. The last case requires exactly i instances, resulting in less elasticity to the developer. Relaxed models with a high relaxation index are preferred.

In addition to the quality characteristics of the models, the time it took to repair the models was also measured for each experiment. When the algorithm is used to repair a model in a development process, the time taken to produce a repaired version should be as low as possible in order to let the developer spend the time productively.

When a new experiment started with new preferences, the knowledge gained from the previous experiment was deleted to make sure the results did not affect each other. All the experiments returned the solution with the highest weight, and this is the chosen model on which to calculate the quality characteristics. Other potential solutions were discarded.

The different combinations of preferences were:

- Prefer to repair higher in the context hierarchy.
- Prefer shorter sequences of actions and prefer to repair higher in the context hierarchy.
- Punish deletion.
- Prefer longer sequences of actions and punish modification of the original model.
- Prefer to repair higher in the context hierarchy and punish deletion.
- Prefer shorter sequences of actions, punish deletion and punish modification of the original model.
- Prefer longer sequences of actions, prefer to repair lower in the context hierarchy and reward modification of the original model.

All of the quality characteristics explained above are numeric, and can be seen plotted in figure 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6 for the ACMR-set. The data is plotted as a line chart because of its ability to visualize trends, and this will show if any selection of preferences clearly performs better or worse than the others overall. If this is the case, a majority of the line representing the preference results will be higher or lower than the other lines.

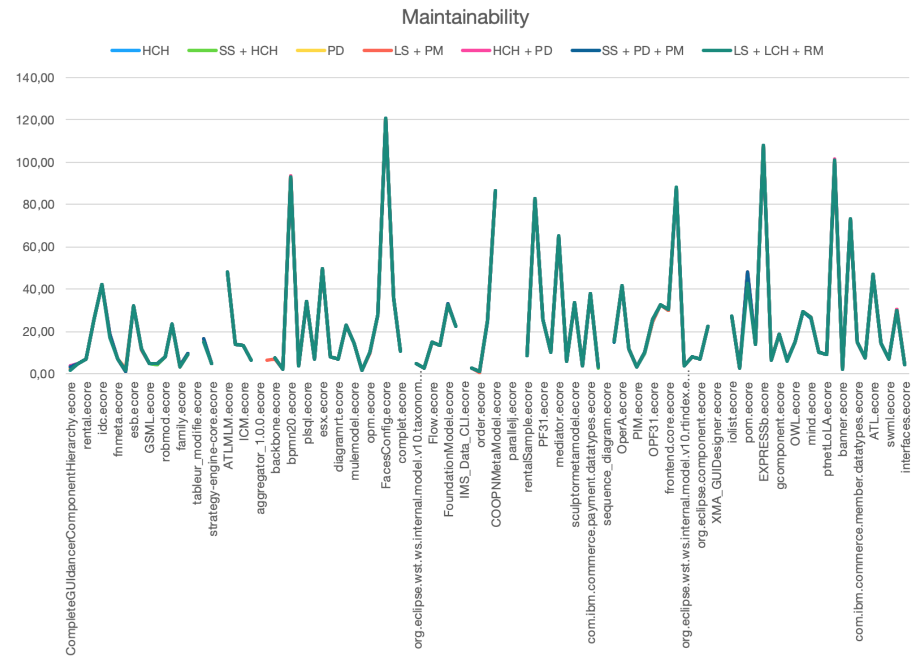

Figure 5.1: A graph showing the difference in maintainability between the results of different settings. The different preference IDs are explained in table 5.2.

The analysis of the results shows very little variation in the data, as all the lines in the plots are placed almost exactly on top of each other. This results in the line primarily being green. The breaks in the graph are solution models in which

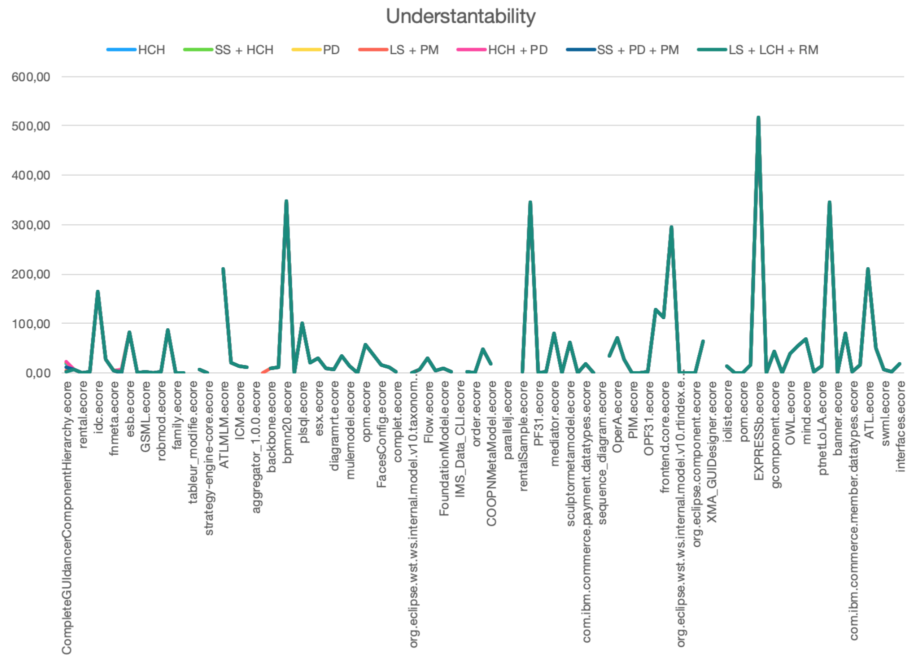

Figure 5.2: A graph showing the difference in understandability between the results of different settings. The different preference IDs are explained in table 5.2.

the metric calculation tool was not able to calculate the quality characteristics, because it was not able to count the number of classes etc. A closer look revealed that these models contained several instances of error 4 (an unsupported error included in the dataset), that might be related to the calculation tool not working.

One outlier in the results is the eclipsecon.ecore-model (hidden label between accregator_1.0.1.ecore and backbone.ecore in the plot) resulting in an orange line. Although all the preferences result in a valid repair for this model, the orange is the only one that the quality characteristics tool can handle. It is uncertain why this is, but the model does initially contain several of the error 4.

Acronym	**Explanation**
SS	Prefer **s**horter **s**equences of actions
LS	Prefer **l**onger **s**equences of actions
HCH	Prefer to repair **h**igher in the **c**ontext **h**ierarchy
LCH	Prefer to repair **l**ower in the **c**ontext **h**ierarchy
PD	**P**unish **d**eletion
PM	**P**unish **m**odification of the original model
RM	**R**eward **m**odification of the original model

Table 5.2: Explanation of the preference ID's.

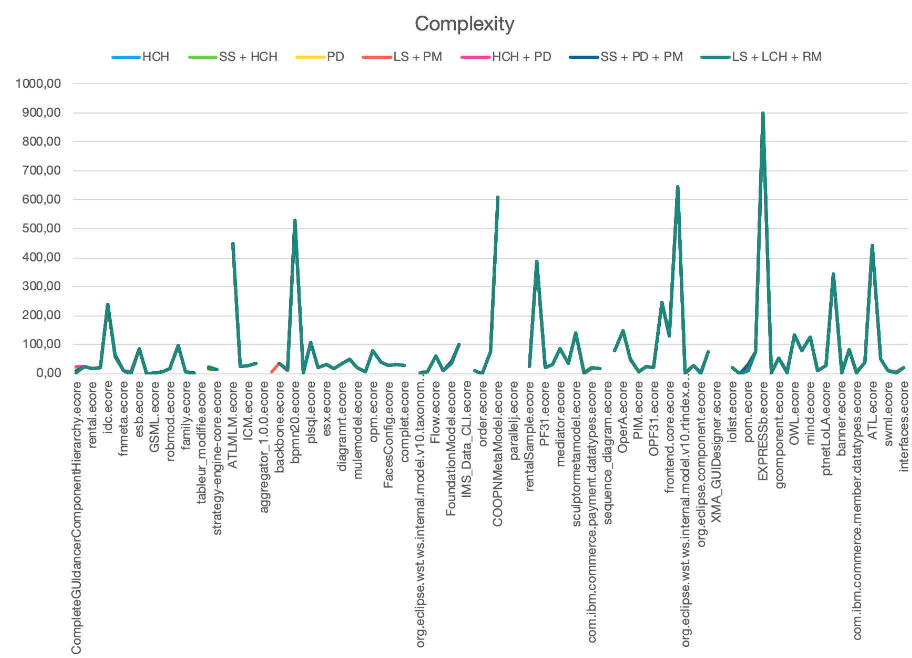

Figure 5.3: A graph showing the difference in complexity between the results of different settings. The different preference IDs are explained in table 5.2.

The orange solution was the only one to select the delete-action twice for this model, that might have resulted in some of these disappearing.

In the relaxation plot (fig. 5.5), there is several values for the pom.ecore. This model contained several of error 50 (A containment or bidirectional reference must be unique if its upper bound is different from 1, see fig. B.11). Some of the preference combinations chose more delete actions than others. For instance, the green line (LS + LCH + RM) choose 27 delete actions for the error 50, whilst the yellow line (PD) chose none. Because the relaxation index is based on adding up the reference bounds difference, this will naturally become lower when a lot of references are deleted.

The time plot shows that most models were fixed very fast, but there are some peeks. In these, there is a clear difference between the faster and the slower settings. The pom.ecore-file was the highest peek at just above 35 minutes (2109571 ms) for the green line (LS + LCH + RM), whilst the yellow (PD) finished in 2 minutes and 23 seconds (143320 ms). Like the relaxation index, this is related to the number of delete-actions. The logs from the execution shows that every time a delete-action was chosen it took a longer time to apply to the model than other actions (often 2-3 seconds). In addition to the 27 delete actions taken in the solution with the highest weight, several others were tried out in the other episodes. As a result, the preferences that try to avoid deletion becomes faster. The other big peak in the time plot is SVG.ecore, and this is also the result of delete actions.

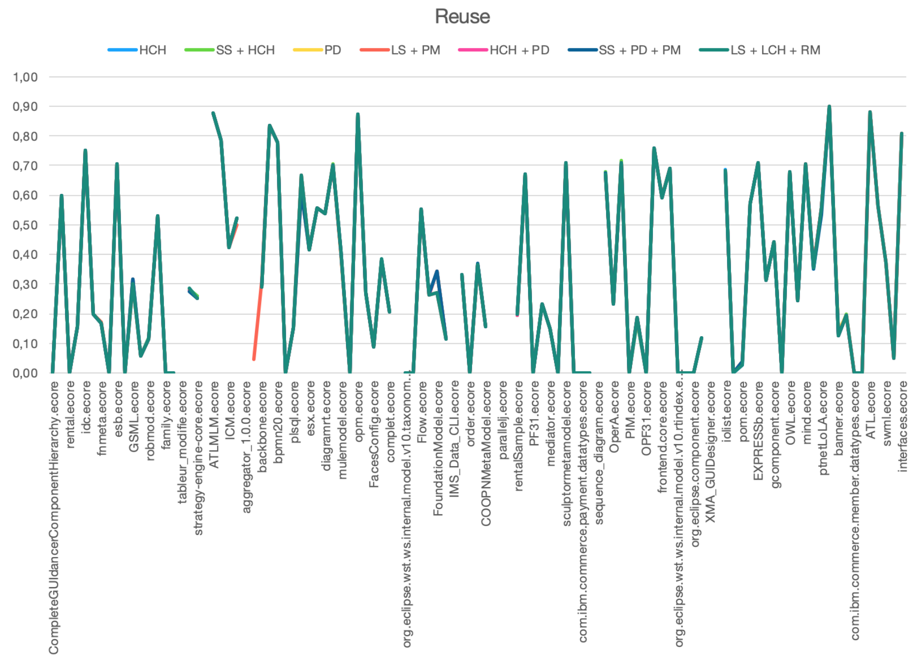

Figure 5.4: A graph showing the difference in the reuse of the metamodel between the results of different settings. The different preference IDs are explained in table 5.2.

The same experiment was conducted on the AMOR set, and the results[2] match with the findings from the ACMR set.

5.2 Can the preferences be customized?

RQ2A questions how new preferences can be added to the algorithm, which is explained in section 3.9. To find out how well this works in practice a hypothesis was put forward, stating that someone other than the author could implement other preferences with only the extensibility-section to guide the development. The success of this experiment was based on whether or not the preferences were implemented correctly combined with feedback from the developer that implemented them.

The developer was one of the authors of [8, 9, 10] and was already familiar with how Parmorel works. However, its structure had changed so much during the work of this thesis that the developer did not recognize the part relating to preferences. As a result, the developer had to follow the content of section 3.9 and look at the existing preferences in order to understand how to implement the new ones.

[2]Available on GitHub: `https://github.com/MagMar94/ParmorelExperimentResults/tree/master/Experiments/Experimen-compare-preferences`

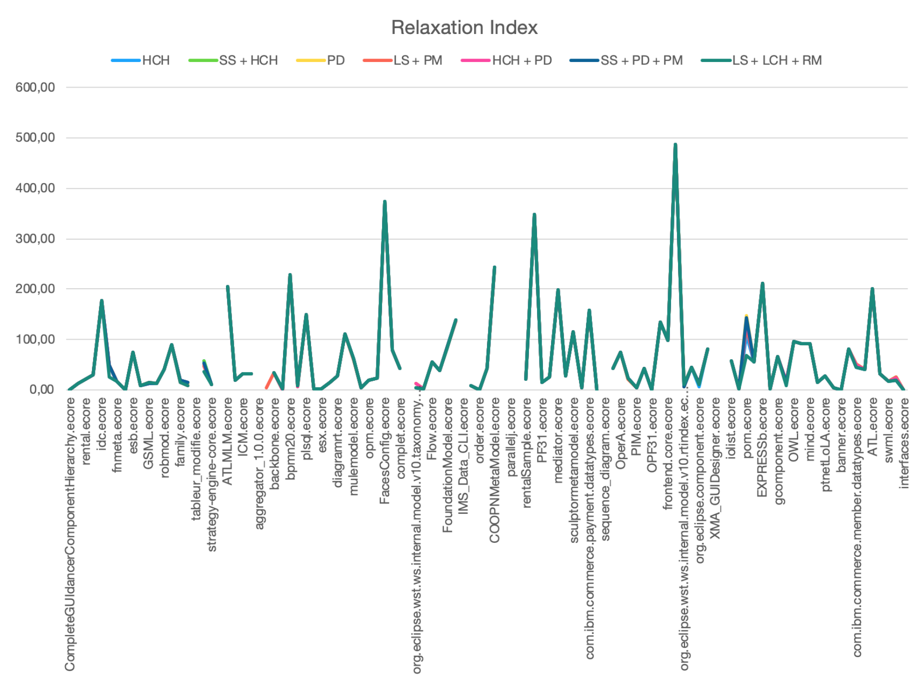

Figure 5.5: A graph showing the difference in the relaxation index of the meta-model between the results of different settings. The different preference IDs are explained in table 5.2.

The following preferences were added to the algorithm:

- Prefer complexity
- Prefer maintainability
- Prefer relaxation
- Prefer reuse
- Prefer understandability

These preferences are based on the quality characteristics described in section 5.1, and each preference works to optimize one of them.

The developer successfully managed to implement all the preferences without guidance and characterized it as quite easy to integrate them. Eclipse dependencies led to some problems, but that was possible to overcome. The file describing the project set-up in the repository was updated to make the Eclipse dependency-issues easier for other developers.

5.3 What is the effect of the model distance?

Working with the new preferences added as a result of the experiment in section 5.2, a new experiment with regards to RQ1 was created. Given that quality

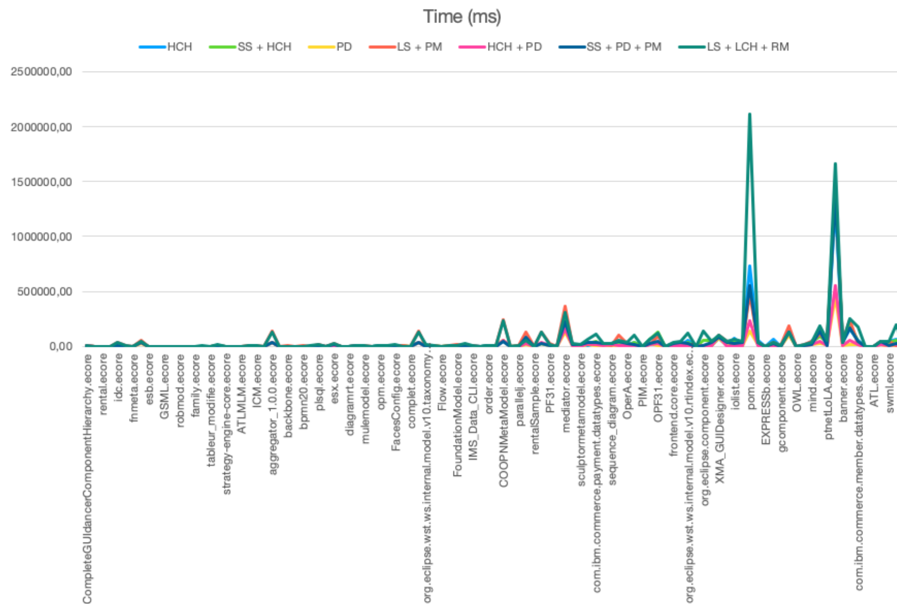

Figure 5.6: A graph showing the difference in execution time between the results of different settings. The different preference IDs are explained in table 5.2.

characteristics are linked to model elements [26], Parmorel should produce repaired updates which optimizes quality characteristics for the model elements impacted by the error. An example is a model containing two classes with the same name. If one of these classes are involved in a hierarchy, the repair can impact all characteristics using the number of hierarchies in their calculation (e.g. understandability). The hypothesis was that altering the distance function used by the distance preference (section 3.10) in combination with preferences trying to optimize the quality characteristics can improve the quality of the resulting solution. By making the distance algorithm stricter considering more aspects of the model, actions affecting properties that was not considered before will equate to the actions that were in terms of distance. As a result, the distance preference will have less of an influence leaving more of the work to the quality characteristic preferences. The findings of this experiment have been submitted as a scientific paper [11] to the technical track of the Models 2020 conference[3].

In order to test this hypothesis, the distance algorithm was altered to take into account the *unique* and *containment* properties. As stated in section 3.10, this was not the case before. Note that this is a tiny modification to the distance algorithm that will make all the combinations listed in table 3.8 getting a distance of 89, because the matching algorithm now will discover that the model elements are different to the original. This modified distance algorithm is henceforth referred to as the *custom* distance. The repair process was executed on the ACMR-set with a combination of the quality characteristic preferences from section 5.2 and the closest distance preference.

[3] `https://conf.researchr.org/track/models-2020/models-2020-technical-track`

model	maintainability	
	original	custom
abapobj.ecore	16.6	16.6
com.ibm.commerce.foundation.datatypes.ecore	30.2	29.8
com.ibm.commerce.member.datatypes.ecore	14.8	14.8
com.ibm.commerce.payment.datatypes.ecore	37.8	37.8
componentCore.ecore	6.6	6.6
ddic.ecore	18.4	18.6
FacesConfig.ecore	120.6	120.4
ICM.ecore	13.2	13.2
org.eclipse.component.api.ecore	9.4	9.4
org.eclipse.component.ecore	7	6.8
org.eclipse.wst.ws.internal.model.v10.registry.ecore	4	4
org.eclipse.wst.ws.internal.model.v10.rtindex.ecore	3.8	3.4
org.eclipse.wst.ws.internal.model.v10.taxonomy.ecore	4.8	4.6
org.eclipse.wst.ws.internal.model.v10.uddiregistry.ecore	4	4
pom.ecore	47.8	45.2
RandL.ecore	22.4	22.4
rom.ecore	10	10
XBNF.ecore	11.8	11.6
XBNFwithCardinality.ecore	2	2

Table 5.3: Model maintainability optimized with custom distance calculator.

As noted in section 3.10, error 50 stating that a containment or bidirectional reference must be unique if its upper bound is different from 1, resulted in different distances because the *unique* and *containment* properties were not considered. The 20 models in the ACMR-set containing this error is listed in table 5.3, 5.4 and 5.5, and the quality characteristics either improved (green), worsened (red) or remained unchanged. The reusability and understandability are not shown in the tables, as the values did not change with respect to the 20 models. In each table, the preference used is the one trying to optimize the quality characteristic in question combined with the distance. Note that this means the resulting characteristics in the different tables are not from the same solution for the given model.

Looking at the entire ACMR-set, the evolution of the quality characteristics with respect to the custom distance calculation can be seen in table 5.6.

5.4 Can the algorithm handle other model types?

RQ2B questions how the algorithm can be adapted to handle other model types. A lot of work has been done to make the algorithm extensible (explained in section 3.9) in order for this to be possible. The hypothesis is that the algorithm can handle other model types as long as it is supplied with the errors in a model and the available actions. If no such model type is found the hypothesis would be rejected. It would have been very interesting to test this hypothesis by implementing support for another model type, but unfortunately, there was not

	complexity	
model	**original**	**custom**
abapobj.ecore	8.54	8.54
com.ibm.commerce.foundation.datatypes.ecore	1.06	1.06
com.ibm.commerce.member.datatypes.ecore	1.22	1.22
com.ibm.commerce.payment.datatypes.ecore	1.44	1.44
componentCore.ecore	6	5
ddic.ecore	36.4	34.4
FacesConfig.ecore	12.12	12.12
ICM.ecore	15.76	15.76
org.eclipse.component.api.ecore	3	1
org.eclipse.component.ecore	3	1
org.eclipse.wst.ws.internal.model.v10.registry.ecore	3	1
org.eclipse.wst.ws.internal.model.v10.rtindex.ecore	3	1
org.eclipse.wst.ws.internal.model.v10.taxonomy.ecore	3	1
org.eclipse.wst.ws.internal.model.v10.uddiregistry.ecore	3.33	1.33
pom.ecore	19.13	15.03
RandL.ecore	99.13	97.13
rom.ecore	25.2	24.2
XBNF.ecore	24.13	22.13
XBNFwithCardinality.ecore	2.83	2.83

Table 5.4: Model complexity optimized with custom distance calculator.

	relaxation index	
model	**original**	**custom**
abapobj.ecore	57	57
com.ibm.commerce.foundation.datatypes.ecore	25	25
com.ibm.commerce.member.datatypes.ecore	51	51
com.ibm.commerce.payment.datatypes.ecore	157	157
componentCore.ecore	17	17
ddic.ecore	47	45
FacesConfig.ecore	374	371
ICM.ecore	32	32
org.eclipse.component.api.ecore	14	14
org.eclipse.component.ecore	12	12
org.eclipse.wst.ws.internal.model.v10.registry.ecore	4.5	5
org.eclipse.wst.ws.internal.model.v10.rtindex.ecore	11	11
org.eclipse.wst.ws.internal.model.v10.taxonomy.ecore	13	13
org.eclipse.wst.ws.internal.model.v10.uddiregistry.ecore	8	8
pom.ecore	148	126
RandL.ecore	139	139
rom.ecore	43	43
XBNF.ecore	24	24
XBNFwithCardinality.ecore	1	1

Table 5.5: Model relaxation index optimized with custom distance calculator.

Quality Characteristic	Improved	Unchanged	Worsened
Complexity	14.46%	81.93%	3.61%
Maintainability	12.05%	78.31%	9.64%
Relaxation index	2.53%	89.87%	7.59%
Reusability	8.64%	83.95%	7.41%
Understandability	2.41%	97.59%	0.00%

Table 5.6: Percentage of models that improved, worsened or remained unchanged with respect to the quality characteristics after applying custom distance.

enough time before the due date of this thesis. The expected result is that the algorithm could be made to handle another type of model because the algorithm itself has no dependencies on EMF.

Chapter 6

Discussion

In this chapter, various aspects of the Parmorel framework and the work done in this thesis are discussed. First, we examine the results from chapter 5. Then we present a review of some of the challenges faced in the thesis.

6.1 Analysis results

Looking at the experiment conducted in section 5.1 there is some variation in the quality characteristics. However, these are usually very small and hardly noticeable in the graphs. Thus, this contradicts the hypothesis that "preferences significantly affect the quality of the repaired models". The algorithm results in models with similar quality regardless of the preferences. One reason could be that a majority of actions do not affect the attributes listed in table 5.1, hence the numbers will stay the same, and in turn provide the same value when calculating quality based on them. Another reason might be that the actions taken do not affect the quality characteristics considerably because the affected elements only constitute a small portion of the model. As a result, the original structure of the faulty model has the largest impact on the characteristics and the repairing actions only introduces minor variations. However, the preferences have a greater impact on the execution time than the quality characteristics. This could be because some actions take longer to perform than others. Which actions are chosen are affected by the preferences, and as a result, they also affect the repair time. In addition, preferences affect how many actions are applied to the model. This is either done directly by rewarding shorter or longer action sequences, or indirectly by for example preferring to repair higher in the context hierarchy (e.g. deleting an entire class in one step instead of deleting several of its attributes over several steps).

Regarding the experiment conducted in section 5.2, the developer managed to implement new preferences relating to completely different aspects of a model than the previously defined ones. A threat to the validity of the experiment could be that developer was already familiar with how the algorithm works.

However, it is expected that developers integrating their own preferences have a better understanding of how the algorithm works than an ordinary user.

The experiment in section 5.3 did affect the quality metrics, as shown in table 5.6. Some of them improved with the stricter distance calculation but remember that the distance only got customized with respect to the matching of references. As a result, only the quality characteristics calculated with references as a factor should improve. The unimproved cases depend on what model elements are involved in the calculation of the quality characteristics, and what errors are in the model. There is also a random element to the Q-learning that can affect the outcome. Different executions will get slightly different results and some quality characteristics could be more or less favored.

In order to get a better understanding of the experiment conducted in section 5.3, the quality characteristics must be understood. Maintainability is defined by the number of structural features, hierarchies, and reference siblings in formulae 5.1, and complexity is defined in terms of static relations between classes (i.e. number of references) in formulae 5.3. Looking at the formulas, both can be improved by lowering the number of references. Removing a reference will, as detailed in table 3.8, increase the distance to the original model. Hence, the algorithm will prefer to change the unique or containment properties when running with the distance preference. However, the custom distance measure detects changes in these properties and makes all the action combinations listen in table 3.8 register a distance of 89. Deleting a reference will now provide a model equally close to the original as making changes to the containment or unique properties. This should result in the model selecting delete actions more often, improving complexity and maintainability. Indeed, the characteristics improve as can be seen in table 5.6. The references play a larger role in the complexity than the maintainability, as can be seen in the formulas. Hence it makes sense that the complexity performs better. Looking at the models that did not improve, error 38 is the most prominent occurrence (example in fig. B.8). This error reports invalidly specified literals and should not be affected by the different distance algorithms.

Understandability (defined in formulae 5.2) and reusability (defined in formulae 5.4) should not improve with respect to the custom distance. They are calculated based on the number of predecessors, classes and features. Table 5.6 shows a very balanced effect on the reusability, with a similar number of improved and worsened models. This is probably down to the randomness of the Q-learning algorithm or individual specific feature differences in each model. Understandability has actually improved slightly in two models, but the errors in these models are regarding names that are not well formed. The improvement is likely the result of the Q-learning randomness.

Lastly, the relaxation index defined in formulae 5.5 is the only quality characteristic to be noticeably harmed by the custom distance. This characteristic indicates how *relaxed* references are with respect to bounds. Cardinality constraints set to, e.g., [1..5] are more relaxed than constraints set to [1..1]. As with the maintainability and complexity characteristics, the original distance preferred to make changes to the unique and containment properties. However, the custom distance makes changes to these properties equal to deleting the reference or changing the upper bound. As a result, changing the upper bound

will get chosen more frequently by the algorithm running with the distance preference. In section 3.5 it was pointed out that actions requiring integer parameters, like `setUpperBound`, always will get the parameter 1. As a result, the upper bound always becomes 1, which decreases the relaxation. The possibility to provide better parameters, possibly based on the error messages using ML, are discussed further in section 9.

Both of the experiments conducted in sections 5.1 and 5.3 use datasets. The datasets are constructed from existing projects on GitHub, but they differ in how the errors are introduced to them. The ACMR-set is taken directly from GitHub without any alteration and consists of 107 models, whilst the AMOR-set consists of 100 models that have been mutated from five consistent models. Although the datasets could be considered small, this threat may be mitigated with the heterogeneity of the sources; these models have been retrieved from different GitHub repositories and hence from different modelers. Likewise, errors in the dataset are of different nature, organic and mutated. It would be interesting to see if larger datasets or multiple runs would give the same results.

6.2 Infrastructure challenges

The tool developed in this thesis is the first Eclipse Plug-in created by the author, as well as the first project dealing with the EMF. Naturally, this led to several challenges to overcome.

The first obstacle was poor documentation. Eclipse does provide some documentation for classes and methods for the EMF API, but this was not always sufficient. One example is the `EditionDistance`-class from `EMF Compare` mentioned in section 3.10. A lot of time was spent trying to make this class calculate a distance between two models, but the lack of documentation made it difficult. This resulted in a post in the Eclipse Community Forum, before aborting it altogether when the community stated this was not the designed use case for the distance calculation [22].

One issue encountered early was the way Eclipse handles dependencies. The project relies on packages provided by Eclipse plug-ins, but they were not always easy to find and import. The code was separated into several Git Submodules, and different modules had different dependencies. Since the code was nested, one might expect the encapsulating project to automatically get the same dependencies as the encapsulated project. However, this had to be done manually. Some modules had a manifest file where the dependencies could be listed, whilst others did not. At first, this was resolved by finding the matching jar-file and adding it to the project, but later the projects were converted so they got a manifest-file. The project could possibly benefit from a project management tool like Maven or Gradle.

Several classes were altered when refactoring the code. The changes were incremental, and in between changes, the original repair script was used to make sure the code still worked. When classes contained in the Q-table were changed or deleted the program failed. The exception message referred to altered or deleted classes. This was confusing, as old classes should not affect new executions. The

reason for this error had to do with the way the Q-table was stored. It serialized the actions, and when it tried to load them from the file, the class definition had changed. In turn, the classes could not be instantiated.

Chapter 7

Related Work

Many tools and research efforts for automatically resolving bugs already exist, and these have shown promising results [40]. The capability to learn from each repaired model and the extensibility the framework provides are the main features that distinguish Parmorel from other model repair approaches.

We could not find any research applying RL to model repair, nor any approaches providing Parmorel's degree of customization. The closest alternative found is Badger, a tool proposed by [48] that uses a recursive best-first search to generate repair plans from an inconsistent model to a consistent one. The user can attach costs to actions and select the desired repair plan but allowing the user more control through customization of the cost function is left for future work. Parmorel grants the user more control by specifying custom preferences if the existing ones are not satisfactory. Another difference is that RL allows Parmorel to perform better after each execution.

One model repair approach is a technique that repairs process models with respect to a behavior log [24]. The implementation focuses on the *fitness* of process models, i.e. the observed behavior in the logs can be explained by the model. If this is not the case, a repair is performed on the unfitting model. By aligning the model and the log, the unfitting events in the log are identified. These are decomposed into several sublogs, and for each sublog there is either discovered a loop that can replay the sublog or a subprocess is derived and added to the repaired model. In other words, the technique both discovers lacking conformance between the log and the model and fixes it. This differs from our solution, as our algorithm needs to have the inconsistencies reported. However, our solution does not require a behavior log and can handle any error if provided with the appropriate actions.

Another approach using behavior logs is proposed by [7]. The authors claim that existing repair methods add too much behavior to the model as a result of their autonomy and results in over-generalized solutions. Instead, they present an approach that shows the deviations from the log to the user along with visual guidance on how the model can be repaired. The user is then responsible for incrementally deciding which discrepancies to fix and how. Parmorel provides more autonomy but counters the over-generalized solutions by letting the user

select a satisfactory repair. This combined with the visual comparison between the solution and the original model allows the user to see where the model has changed and make potential necessary changes after the repair has finished (e.g. change attribute names). It might be easier for the user to make the required alterations when the complete solution is presented, than during the repair process when the outcome is unknown.

An Eclipse-plug-in is proposed by [43]. This approach repairs models with a rule-based approach but can take input from the user in an interactive manner. In this solution, the repair process follows specified steps in a certain order, but the user can be consulted every time a decision has to be made. Typical decisions could be what node to delete to avoid exceeding an upper-bound invariant or what type of node to create if a node is missing. The algorithm can alternatively make these decisions randomly. This differs from our solution in that it is heavily dependent on the user during the repair process (or not dependent at all if the decisions are made randomly), whilst our solution provides more autonomy. Our technique repairs the model with respect to the user requirements provided to the algorithm before the repair process starts, and the algorithm does not consult the user whilst repairing.

ReVision is another proposed interactive tool implemented on top of the Eclipse modeling technology stack [46]. It exploits information from the editing history of a model containing information about where the defect originated. It searches all the applied operations and looks for the corresponding change in the model. If the operation leads to an inconsistency, the tool searches for a complementing repair. This is then given to the user in form of recommendations, meaning the algorithm does not make any decision. Parmorel, on the other hand, will execute the repair action directly providing the user with the final repaired solution. Another difference is that ReVision is dependent on the models editing history.

Other efforts have been made to improve the modeling experience that do not address the repair of bugs. One example is a recommender system with proactive modeling that has been proposed in the form of a plug-in [41]. Proactive modelling is a technique that anticipate model transformations and executes them automatically, only consulting the modeler when necessary. This works until the algorithm is faced with non-deterministic modeling actions like add, delete or edit. At this point the modeler has to tell the engine how to progress, but this can be a challenging task in large and complex Domain Specific Modelling Languages. This is where the recommender system helps the modeler by providing appropriate actions. The proposed solution uses machine learning in the form of ensemble learning, i.e. multiple classifiers or learning algorithms combined into one. The classifiers are built for different recommendation parameters, and weights can be assigned by the user to select parameters to emphasize their subjective importance. The parameters tell something about the action with regards to the its context and history. For example, one parameter specifies how recently a modeling action was chosen, because a recently chosen action is likely to be chosen again. In comparison, Parmorel do not have the historical information regarding the actions. However, the Parmorel preferences can be extended to handle anything from action specific rewards to end result quality. Other differences between the algorithms is the autonomy Parmorel provides, whilst the recommender system suggest operations to the user that applies them

manually. This is a result of the different use cases the solutions targets.

Another effort by [49] is working on improving how developers deal with the complexity of software modeling by implementing a software agent using artificial intelligence. This software agent will be embedded in an IDE and help the developer to create models by catering for new ideas, providing recommendations and help. The implementation of software agents requires a knowledge repository, which also will be provided by the authors. This is another way of using machine learning in dealing with MDSE that does not focus on the handling of errors. Although the agent might be able to handle errors in the model, no such claims have yet been made. Another difference to Parmorel is the need for a knowledge repository on which to train the agent. Parmorel uses RL and thus do not require training data, making it easier to adapt to other model types and environments.

Chapter 8

Conclusion

Models are central objects in MDSE, and errors might be introduced to them in the development process. Model repair solutions can help developers deal with these inconsistencies. The ACMR-set introduced in chapter 5 showed that 49% of the models found by [44] contained errors, which indicates the need for such tools. In this thesis, an extensible plug-in has been created that provides automatic model repair based on personal preferences. By making the underlying algorithm extensible, a framework has been produced with the possibility of implementing support for other model types and new preference options.

RQ1 questioned how much personal preferences affected the solutions. The experiment conducted in section 5.1 tried to measure this on maintainability, understandability, complexity, reusability, and relaxation index. Although the numbers were not significantly affected by the preferences, this proves that the algorithm performs equally well with respect to the quality characteristics regardless of the preferences. In order to see the impact of one specific preference, the experiment in section 5.3 compared an execution with a slightly altered preference to the original and noted a clear difference. Indeed, the quality characteristics were affected by the preferences. This in spite of the change made to the preferences was very small, only considering two more properties of the reference attributes. The preferences do not have a huge impact on the final quality of the model, as only the portions of the model containing errors will be affected. However, the proposed solutions are heavily dependent on the preferences. As a result, the conclusion is that personal preferences have a significant impact on the proposed solutions.

RQ2A asked how new preferences could be added to the algorithm. The work done in this thesis has made it possible to supply Parmorel with new preferences by extending interfaces and implementing an abstract class as explained in section 3.9. This was verified by the experiment conducted in section 5.2, in which a developer unfamiliar with the new layout of the preferences established new ways of rewarding actions. The new preferences were based on quality characteristics, an aspect of the model previously not considered by the algorithm. This indicates that the algorithm can be influenced by any model aspect which

has a numerical representation.

RQ2B asked how the application can be reengineered, and what adaptations are required in order for the algorithm to handle other model types. Section 3.9 explains how the work done in this thesis has made it possible to extend the algorithm to handle other model types and even other approaches to model repair. As explained in section 5.4 this was, unfortunately, never tested in the thesis. However, because no classes outside the `no.hvl.projectparmorel.qlearning.ecore`-package import from the Ecore-library, and the repair procedure is done outside of this package, the algorithm is not dependent on EMF. As a result, only the constraints introduced by the abstract classes and interfaces in the Parmorel framework limits what model types can be implemented.

Chapter 9

Further Work

Although the code has been through several revisions, there is still lots of work that should be done in order to improve the quality. Some classes are big and can quickly become unmanageable, especially for developers new to the project. One such class is `QModelFixer`. It would be interesting to see if this class and some of the contained methods could be made smaller and more expressive. One way to do this might be to make the episodes and the steps into separate classes where the logic can be placed.

The testing also needs improvement. As of now, very few unit tests exists and the result of the integration tests are not automatically verified. This can make it hard to refactor the code, as it cannot quickly be proven to still work like it should. Adding more automatic test cases would be good when improving the code quality, but also for helping implementing support for new metamodels etc. If test cases are written for the abstract classes and interfaces, the implementation can quickly be verified to function in accordance with the specification.

The error messages are currently ignored by Parmorel, which only focuses on the error codes. Additionally, the parameters provided to the actions are fixed. This is an interesting area to explore even further, considering the error messages often provide a better description of the models' state. Error messages combined with a more advanced parameter selection using ML could prove powerful, and potential new solutions can be found. An example where this could be beneficial is for EMF error number 39, "The lower bound -1 must be greater than or equal to 0". If the algorithm could read the error message, the algorithm might try to set the lower bound to a valid number (0 or above). As of now, all integer parameters are fed the number 1 regardless of the message.

Another task that could be very beneficial is extending the algorithm, making it able to handle models derived from other metamodels. A theoretical explanation on how this can be achieved is provided in section 3.9, but once the implementation starts one might find more or less similarity than anticipated between the model types. This can again lead to more abstractions being required.

Currently, the weights used by the Q-learning are decided through experiments

figuring out what worked and what did not. These weights could alternatively be set by the user, but that would require a greater understanding of the algorithm. Instead, the default weights could be influenced by the user. Rather than selecting the preference *Punish deletion*, the user could select *Punish deletion a little* or *Punish deletion moderately*. A slider could be implemented, providing the user with a visual tool for deciding how much a preference should affect the algorithm.

Unsupported errors were explained in section 3.7. An error is marked as not supported if none of the actions available results in the removal of an error. Allowing users to supply custom actions to the algorithm in addition to what is supplied by the modeling framework would make it possible to handle previously unsupported errors.

An interesting option would be to avoid the users selecting preferences, but rather have them select the solution they like the most to a broken model. The algorithm could randomly choose preferences and repair the model in different ways. After enough iterations, the algorithm would be more likely to present the users with a solution tailored to them. Given this time consuming and tedious work, this would require a training phase prior to deployment. This training phase would still use RL, but most of the rewards would be given at the end of the repairing process when a solution is selected by a developer. Developers could be included in training the algorithm by using the tool whilst they work, providing data to gradually build some knowledge. When enough knowledge is obtained, the tool could be released to benefit all developers. This idea can be expanded even further, by letting the algorithm train itself. A way to achieve this could be to break consistent models and attempt to repair them again, rewarding the solutions closest to the original based on their distance.

Appendix A

Source code

The source code for the plug-in is available at this URL: `https://github.com/MagMar94/ParmorelEclipsePlugin`.

The source code for the underlying Parmorel algorithm is available here: `https://github.com/MagMar94/projectparmorel`.

Appendix B

Error explanations

What follows are explanations of the errors in the ACMR-set introduced in chapter 5 that are supported by Parmorel. The error codes are based on the values reported by EMF.

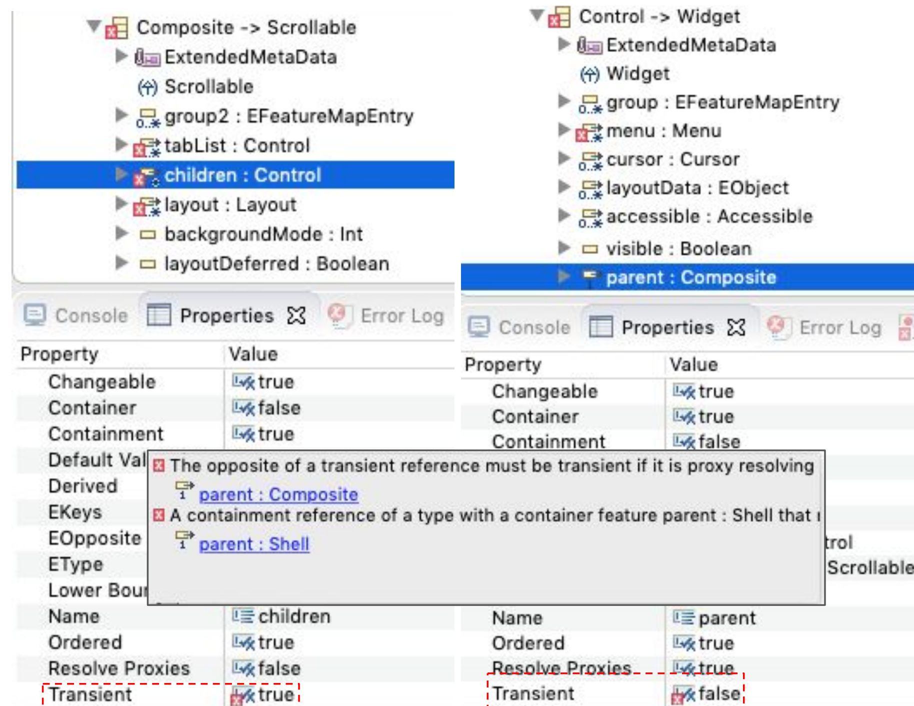

Figure B.1: Error 11 identified in xwt09_updating.ecore

Error 11 is triggered when a transient reference has an opposite reference that is not declared transient. This error only occurs in xwt09_updating.ecore but appears twice in that model. Figure B.1 illustrates the transient reference chil-

dren (left) has the opposite parent in class Control that is set as not transient (right).

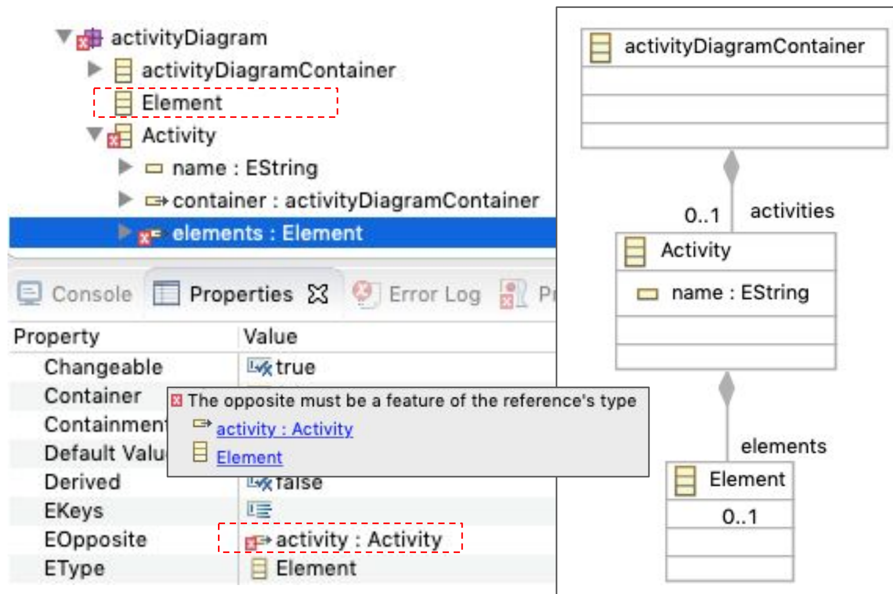

Figure B.2: Error 13 identified in activityDiagram.ecore

Error 13 only appears only once in a model called activityDiagram.ecore and can be seen in fig. B.2. This is related to the fact that a reference, in this case `elements` in the class `Activity` has set the opposite to `activity` in class `Element`. However, the `Element` class seems to be empty. The opposite is incorrectly declared in the class `ControlFlow` instead.

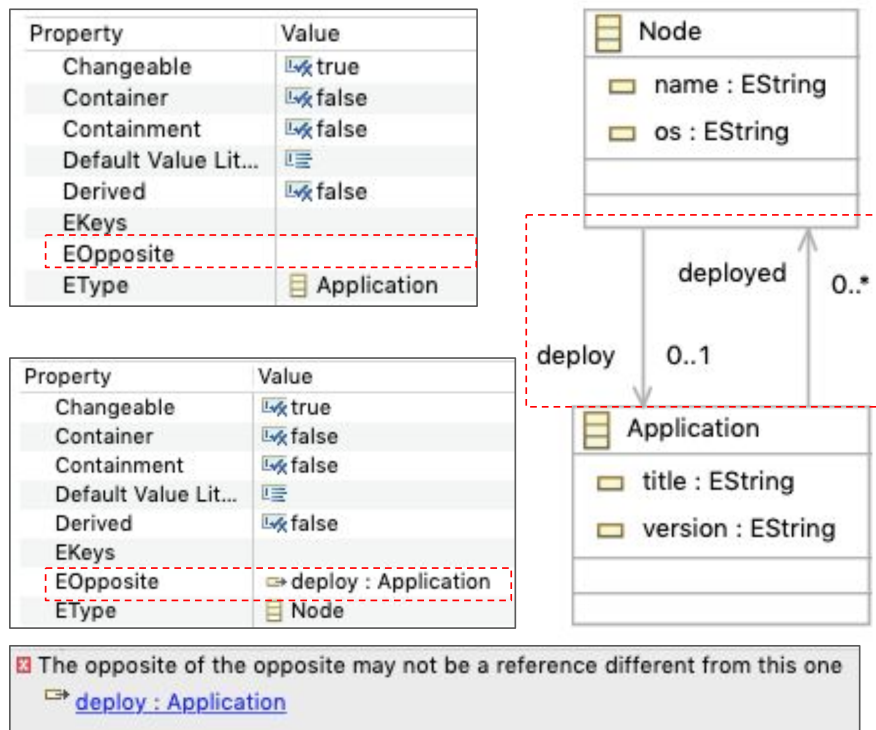

Figure B.3: Error 14 identified in primer.ecore

Error 14 is shown in fig. B.3. In this example, the reference `deployed` (at the bottom) has set the opposite reference to `deploy` but `deploy` (at the top) has not set the opposite. One of the effects is that the diagram shows two references instead of a single one, since one of the references has an unset opposite.

Figure B.4: Error 17 identified in diagramrt.ecore

Error 17 appears 7 times in the dataset. This error states that an attribute not declared as transient must have a datatype that serializable. An example is visualized in fig. B.4. In this case the attribute `domainResource` is not declared transient, but its type `EResource` is not serializable.

PiecewiseLinearCostTable -> CostTable
CostTable
number : EInt
label : EString
numberOfCoordinates : EInt
coordinatePairs : EMap<EFloat, EFloat>
EMap<EFloat, EFloat>

Problems on Children
EFloat
The primitive type 'float' cannot be used in this context
EFloat
The primitive type 'float' cannot be used in this context

Figure B.5: Error 22 identified in OPF31.ecore

Error 22 has 4 occurrences in the data set, with all of them located in OPF31.ecore. This error states that "The primitive used type cannot be used in this context". In this example, the attribute `coordinatePairs` is a Map from `EFloat` to `EFloat`. `EFloat` cannot be used in this context and needs to be replaced it with another type.

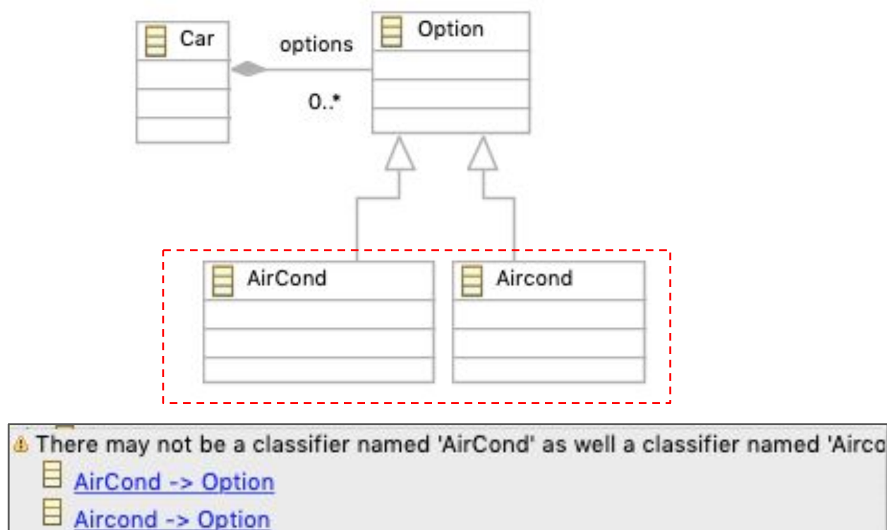

Figure B.6: Error 29 identified in car.ecore

Error 29 states that two or more classifiers have the same name. This error occurs twice in the data set. The example in fig. B.6 highlights two classes `AirCond` and `Aircond` that only differs in letter casing; hence the names are considered equal.

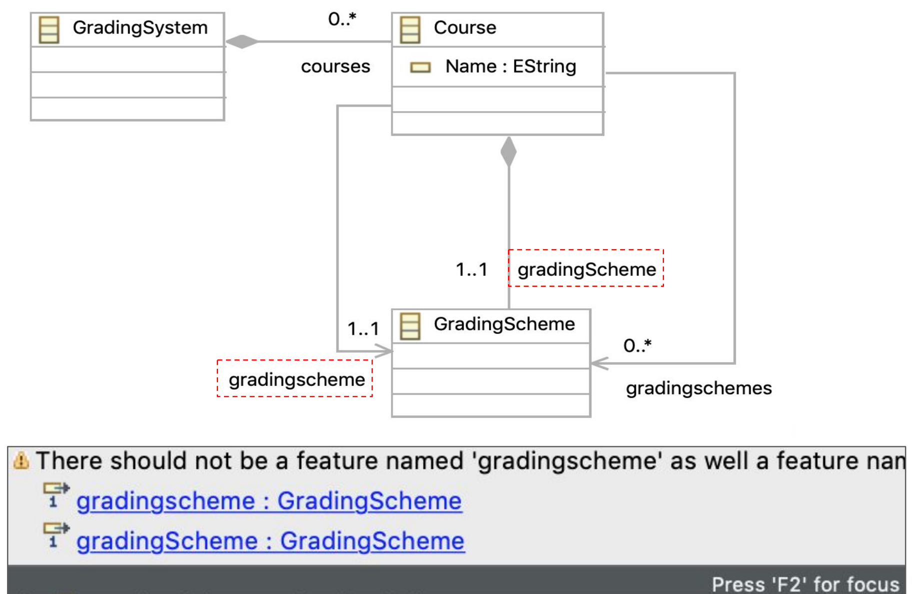

Figure B.7: Error 32 identified in GSML.ecore

Error 32 occurs when two or more feature in the same classifier have been defined with same name, or the names only differ in letter casing. This error has 20 occurrences in the data set. The example illustrated in fig. B.7 shows the class `Course` containing two features where names are considered equal: `gradingscheme` and `gradingScheme`.

Figure B.8: Error 38 identified in OPF31.ecore

Error 38 is a very common error in the data set with 166 occurrences. This error states that the default value specified is not coherent with the literals specified in the enumeration. The example in fig. B.8 shows the attribute `limitType` of type `LimitType` is set to 1, when the literals in the enumeration consists of: Reporting, Hard, SoftLinear, SoftQuadratic.

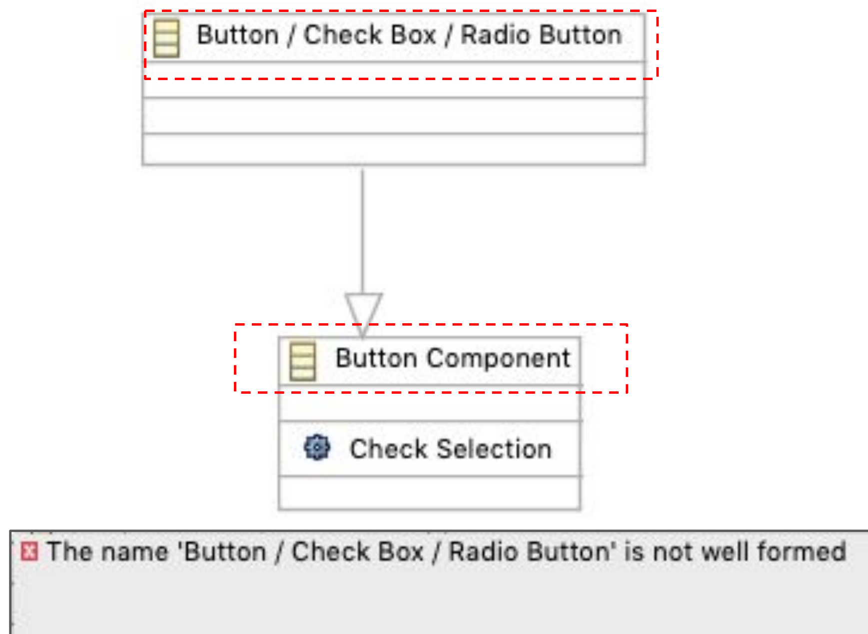

Figure B.9: Error 44 identified in GUIdancerComponentHierarchy.ecore

Error 44 is another diffused error in the data set appearing 216 times, but most of them exist in GUIdancerComponentHierarchy.ecore. This error states that names are not well formed. The example in fig. B.9 highlights two of these errors, where the names contain forbidden characters, e.g., / or empty spaces.

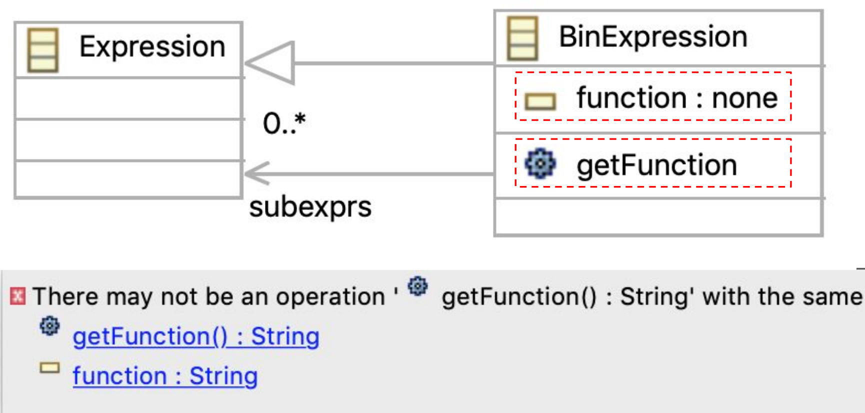

Figure B.10: Error 48 identified in tableur_modifie.ecore

An operation cannot be declared with the same signature as an accessor method for a feature, and this is reported by error 48. The example in fig. B.10 shows a class containing an operation `getFunction()` and a feature named `function`.

This will result in a conflict, because EMF automatically generates getter- and setter-methods for the structural features of the classes. The signature of the generated method will be equal to the defined method, hence the error.

Figure B.11: Error 50 identified in org.eclipse.wst.ws.internal.model.v10.taxonomy.ecore

Error 50 reports that a containment reference must be unique if the upper bound is different from 1. This error exists 160 times in the dataset and is one of the most diffused errors. Fig. B.11 displays an example of this error, where the containment references `xMLNSPrefixMap` and `xSISchemaLocation` are neither declared unique, nor is the upper bound set to 1.

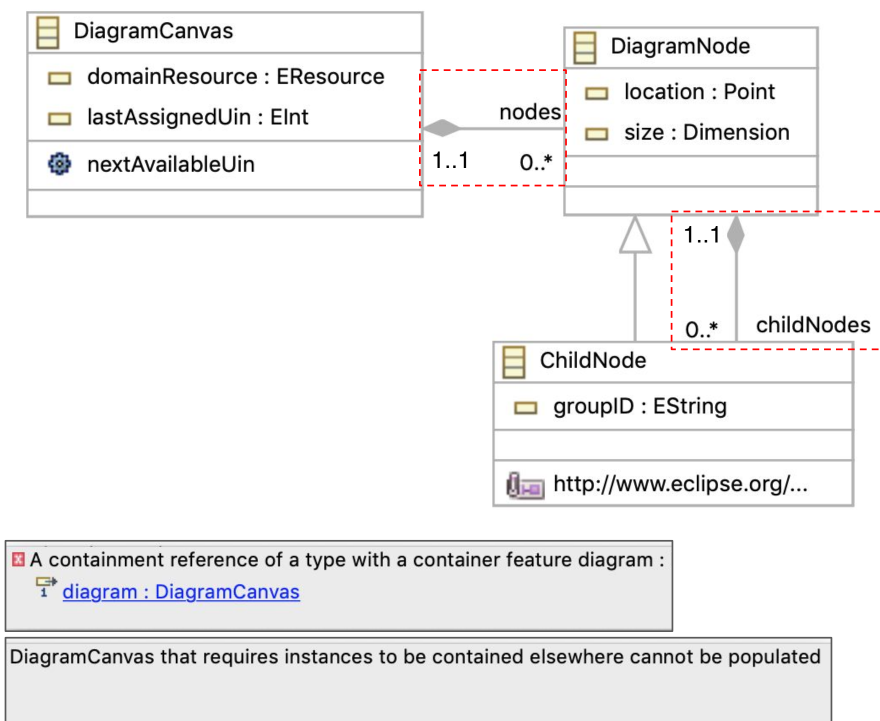

Figure B.12: Error 51 identified in diagramrt.ecore

Error 51 says that a containment reference of a type with a container feature that requires instances to be contained elsewhere cannot be populated. This error occurs 94 times in the data set. Looking at the example in fig. B.12, the class `DiagramCanvas` contains the class `DiagramNode` through the containment reference `nodes`. In turn, `DiagramNode` contains `ChildNode` through another containment reference `childNodes`. The `ChildNode` requires at least one `DiagramNode` to be contained in because of the `childNodes` opposites lower bound of 1, but the `nodes` reference has a lower bound of 0. As a result, a `DiagramCanvas` might not exist for the `ChildNode` to be contained in, and this violates the constraints.

Bibliography

[1] Lorenzo Addazi et al. "Semantic-based Model Matching with EMFCompare." In: *Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016*. Ed. by Tanja Mayerhofer et al. Vol. 1706. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 40–49. URL: http://ceur-ws.org/Vol-1706/paper6.pdf.

[2] Charu C. Aggarwal, ed. *Data Classification: Algorithms and Applications*. CRC Press, 2014. ISBN: 978-1-4665-8674-1. URL: http://www.crcnetbase.com/doi/book/10.1201/b17320.

[3] Ethem Alpaydin. *Introduction to machine learning*. eng. Third. Adaptive computation and machine learning. Cambridge: MIT Press, 2014. ISBN: 9780262028189.

[4] Kerstin Altmanninger et al. "AMOR–towards adaptable model versioning." In: *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*. Vol. 8. 2008, pp. 4–50.

[5] Sanaa Alwidian and Daniel Amyot. "Relaxing metamodels for model family support." In: *11th Workshop on Models and Evolution (ME 2017)*. Vol. 2019. 2017, pp. 60–64.

[6] Thorsten Arendt. *Quality Assurance of Software Models - A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project; Qualitätssicherung von Softwaremodellen - Ein strukturierter Qualitätssicherungsprozess unterstützt durch eine flexible Werkzeugumgebung innerhalb des Eclipse Modeling Project*. eng. 2014.

[7] Abel Armas Cervantes et al. "Interactive and Incremental Business Process Model Repair." In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*. Ed. by Hervé Panetto et al. Cham: Springer International Publishing, 2017, pp. 53–74. ISBN: 978-3-319-69462-7.

[8] Angela Barriga, Adrian Rutle, and Rogardt Heldal. "Automatic model repair using reinforcement learning." In: *MODELS Workshops*. Vol. 2245. CEUR Workshop Proceedings. Copenhagen: CEUR-WS.org, Oct. 2018, pp. 781–786. URL: http://ceur-ws.org/Vol-2245/ammore_paper_1.pdf.

[9] Angela Barriga, Adrian Rutle, and Rogardt Heldal. "Journal of Object Technology." In: *Applying reinforcement learning to personalize automatic model repairing* (2019).

[10] Angela Barriga, Adrian Rutle, and Rogardt Heldal. "Personalized and Automatic Model Repairing using Reinforcement Learning." In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Sept. 2019, pp. 175–181. DOI: 10.1109/MODELS-C.2019.00030.

[11] Angela Barriga et al. "An extensible framework for customizable model repair." In: *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2020, Montreal, Canada, October 18-23, 2020*. IEEE, 2020: Submitted, waiting for review.

[12] Francesco Basciani et al. "A tool-supported approach for assessing the quality of modeling artifacts." In: *Journal of Computer Languages* 51 (2019), pp. 173–192.

[13] Brian Beavis and Ian Dobbs. "STATIC OPTIMIZATION." In: *Optimisation and Stability Theory for Economic Analysis*. Cambridge University Press, 1990, pp. 32–72. DOI: 10.1017/CBO9780511559402.003.

[14] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2011. ISBN: 9781139501903. URL: https://books.google.no/books?id=9u0gAwAAQBAJ.

[15] Michael W. Berry. *Supervised and Unsupervised Learning for Data Science*. eng. Cham, 2020.

[16] Jean Bézivin. "On the unification power of models." In: *Software and Systems Modeling* 4.2 (2005), pp. 171–188. DOI: 10.1007/s10270-005-0079-0. URL: https://doi.org/10.1007/s10270-005-0079-0.

[17] Marco Brambilla. *Model-driven software engineering in practice, second edition*. eng. 2nd ed. Synthesis lectures on software engineering ; 4. S.l.]: Morgan & Claypool Publishers, 2017. ISBN: 1627057080. URL: http://portal.igpublish.com/iglibrary/search/MCPB0006316.html.

[18] Frederick P. Brooks. "No Silver Bullet Essence and Accidents of Software Engineering." In: *Computer* 20.4 (Apr. 1987), pp. 10–19. ISSN: 1558-0814. DOI: 10.1109/MC.1987.1663532.

[19] Cédric Brun and Alfonso Pierantonio. "Model differences in the eclipse modeling framework." In: *The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34.

[20] Horst Bunke. "On a relation between graph edit distance and maximum common subgraph." In: *Pattern Recognition Letters* 18.8 (1997), pp. 689–694.

[21] Field Cady. "Machine Learning Classification." eng. In: *The Data Science Handbook*. New York: John Wiley & Sons, Incorporated, 2017, pp. 97–120. ISBN: 9781119092940.

[22] *Eclipse Community Forum post*. URL: https://www.eclipse.org/forums/index.php/t/1102085/ (visited on May 13, 2020).

[23] *EditionDistance documentation*. URL: https://www.eclipse.org/emf/compare/documentation/latest/developer/javadoc/org/eclipse/emf/compare/match/eobject/EditionDistance.html (visited on Apr. 9, 2020).

[24] Dirk Fahland and Wil M.P van Der Aalst. "Model repair — aligning process models to reality." eng. In: *Information Systems* 47.C (2015), pp. 220–243. ISSN: 0306-4379.

[25] Guojun Gan. "Design Patterns." eng. In: *Data Clustering in C++: An Object-Oriented Approach.* Chapman and Hall/CRC, 2011, pp. 57–77. ISBN: 9781439862247.

[26] Marcela Genero and Mario Piattini. "Empirical validation of measures for class diagram structural complexity through controlled experiments." In: *5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering.* 2001.

[27] Shekhar Gulati. *Java Unit Testing with JUnit 5 : Test Driven Development with JUnit 5.* eng. Berkeley, CA, 2017.

[28] Yo-Sub Han, Sang-Ki Ko, and Kai Salomaa. "Computing the Edit-Distance between a Regular Language and a Context-Free Language." In: *Developments in Language Theory.* Ed. by Hsu-Chun Yen and Oscar H. Ibarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–96. ISBN: 978-3-642-31653-1.

[29] W. Heath Hoagland and Lionel Williamson. "Feasibility studies." In: *Department of Agricultural Economics, the University of Kentucky* (2000).

[30] Jubair J. Al-Ja'Afer and Khair Errin M. Sabri. "Metrics for object oriented design (MOOD) to assess Java programs." In: *King Abdullah II school for information technology, University of Jordan, Jordan* (2004).

[31] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation.* Springer US, 2013. ISBN: 9781475733044. URL: `https://books.google.no/books?id=iJTkBwAAQBAJ`.

[32] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "A rule-based approach to the semantic lifting of model differences in the context of model versioning." In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).* IEEE. 2011, pp. 163–172.

[33] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. "Detecting and exploring side effects when repairing model inconsistencies." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering.* 2019, pp. 113–126.

[34] David G. Kleinbaum and Mitchel Klein. *Logistic Regression.* Third. Statistics for Biology and Health. New York: Springer, 2010. ISBN: 978-1-4419-1742-3. DOI: `https://doi.org/10.1007/978-1-4419-1742-3`.

[35] Liwu Li. *Java : data structures and programming.* eng. Berlin, Germany ; 1998.

[36] Nuno Macedo, Tiago Jorge, and Alcino Cunha. "A Feature-Based Classification of Model Repair Approaches." In: *IEEE Transactions on Software Engineering* 43.7 (July 2017), pp. 615–640. ISSN: 2326-3881. DOI: `10.1109/TSE.2016.2620145`.

[37] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0-13-235088-2.

[38] James Matson. "Cooperative feasibility study guide." In: (2000).

[39] Ali Mili. *Software testing : concepts and operations.* eng. Hoboken, New Jersey, 2015.

[40] Martin Monperrus. "Automatic Software Repair: A Bibliography." eng. In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–24. ISSN: 03600300.

[41] Arvind Nair. "Integrating recommender systems into domain specific modeling tools." MA thesis. Indianapolis, Indiana: Purdue University, 2017.

[42] Issam El Naqa, Ruijiang Li, and Martin J. Murphy. "What is machine learning?" eng. In: *Machine Learning in Radiation Oncology : Theory and Applications*. 1st. Cham: Springer, 2015, pp. 3–11. ISBN: 3-319-18305-2.

[43] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. "Rule-Based Repair of EMF Models: An Automated Interactive Approach." In: *Theory and Practice of Model Transformation*. Ed. by Esther Guerra and Mark van den Brand. Cham: Springer International Publishing, 2017, pp. 171–181. ISBN: 978-3-319-61473-1.

[44] Phuong T Nguyen et al. "Automated Classification of Metamodel Repositories: A Machine Learning Approach." In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2019, pp. 272–282.

[45] *Object documentation*. URL: `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Object.html` (visited on Apr. 22, 2020).

[46] Manuel Ohrndorf et al. "ReVision: A Tool for History-based Model Repair Recommendations." eng. In: *Proceedings of the 40th International Conference on software engineering*. Vol. 137351. ICSE '18. ACM, 2018, pp. 105–108. ISBN: 9781450356633.

[47] Peter Oram. "WordNet: An electronic lexical database. Christiane Fellbaum (Ed.). Cambridge, MA: MIT Press, 1998. Pp. 423." eng. In: *Applied Psycholinguistics* 22.1 (2001), pp. 131–134. ISSN: 01427164. URL: `http://search.proquest.com/docview/200949690/`.

[48] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. "Resolving model inconsistencies using automated regression planning." In: *Software & Systems Modeling* 14.1 (2015), pp. 461–481.

[49] Maxime Savary-Leblanc. "Improving MBSE Tools UX with AI-Empowered Software Assistants." In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*. Ed. by Loli Burgueño et al. IEEE, 2019, pp. 648–652. DOI: `10.1109/MODELS-C.2019.00099`. URL: `https://doi.org/10.1109/MODELS-C.2019.00099`.

[50] Bran Selic. "Model-Driven Development: Its Essence and Opportunities." In: *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. Apr. 2006. DOI: `10.1109/ISORC.2006.54`.

[51] Mary Shaw. "Writing good software engineering research papers." eng. In: *25th International Conference on Software Engineering, 2003. Proceedings*. IEEE, 2003, pp. 726–736. ISBN: 076951877X.

[52] Frederick T. Sheldon and Hong Chung. "Measuring the complexity of class diagrams in reverse engineering." In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.5 (2006), pp. 333–350.

[53] Frederick T. Sheldon, Kshamta Jerath, and Hong Chung. "Metrics for maintainability of class inheritance hierarchies." In: *Journal of Software Maintenance and Evolution: Research and Practice* 14.3 (2002), pp. 147–160.

[54] Alex J. Smola and Bernhard Schölkopf. "A tutorial on support vector regression." In: *Statistics and Computing* 14.3 (Aug. 2004), pp. 199–222. ISSN: 1573-1375. DOI: `10.1023/B:STCO.0000035301.49549.88`. URL: `https://doi.org/10.1023/B:STCO.0000035301.49549.88`.

[55] Diomidis Spinellis. “Git.” In: *IEEE Software* 29.3 (2012), pp. 100–101.

[56] Dave Steinberg et al. *EMF: Eclipse Modeling Framework*. eng. 2nd ed. Eclipse Series. Pearson Education, 2008. ISBN: 9780132702218. URL: https://books.google.no/books?id=sA0zOZuDXhgC.

[57] Klaas-Jan Stol, Michael Goedicke, and Ivar Jacobson. “Introduction to the special section—General Theories of Software Engineering: New advances and implications for research.” In: *Information and Software Technology* 70 (2016), pp. 176–180.

[58] Eugene Syriani, Robert Bill, and Manuel Wimmer. “Domain-specific model distance measures.” In: *Journal of Object Technology* 18.3 (2019), pp. 1–19. ISSN: 16601769.

[59] Gabriele Taentzer et al. “Change-Preserving Model Repair.” In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Marieke Huisman and Julia Rubin. Vol. 10202. Lecture Notes in Computer Science. Springer, 2017, pp. 283–299. DOI: 10.1007/978-3-662-54494-5_16. URL: https://doi.org/10.1007/978-3-662-54494-5%5C_16.

[60] Frank Tsui and Orlando Karam. *Essentials of software engineering*. eng. 2nd ed. Sudbury, Mass: Jones and Bartlett, 2011. ISBN: 9780763785345.

[61] Christopher J. C. H. Watkins and Peter Dayan. “Q-learning.” In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

[62] Jon Whittle, John Hutchinson, and Mark Rouncefield. “The State of Practice in Model-Driven Engineering.” In: *IEEE Software* 31.3 (May 2014), pp. 79–85. ISSN: 0740-7459. DOI: 10.1109/MS.2013.65.