PARMOREL: PERSONALIZED AND AUTOMATIC REPAIR OF MODELS USING REINFORCEMENT LEARNING

Doctoral Dissertation by
Ángela Barriga Rodríguez

Thesis submitted for
the degree of Philosophiae Doctor (PhD)
in
Computer Science:
Software Engineering, Sensor Networks and Engineering Computing

Department of Computer Science,
Electrical Engineering and Mathematical Sciences

Faculty of Engineering and Science

Western Norway University of Applied Sciences

August 13, 2021

Series of dissertation submitted to
the Faculty of Engineering and Science,
Western Norway University of Applied Sciences.

ISSN: 2535-8146
ISBN: 978-82-93677-56-7

Author: Ángela Barriga Rodríguez
Title: PARMOREL: Personalized and automatic repair of models using reinforcement learning

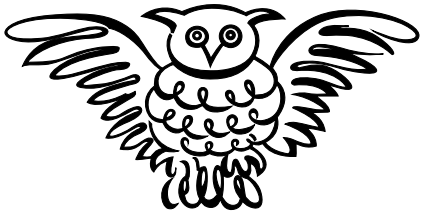
Printed production:

Molvik Grafisk / Western Norway University of Applied Sciences

Bergen, Norway 2021

TO MY FAMILY,

for nurturing me with the love
that allowed me to bloom
into the person I am today.



PREFACE

The author of this thesis has been employed as a Ph.D. research fellow in the software engineering research group at the Department of Computer Science, Electrical Engineering and Mathematical Science at Western Norway University of Applied Sciences. The author has been enrolled in the Ph.D. programme in Computer Science: Software Engineering, Sensor Networks and Engineering Computing. The specialization of this thesis is Software Engineering.

The research presented in this thesis has been accomplished in cooperation with the Western Norway University of Applied Sciences and the University of Málaga, Spain. Likewise, through the collaboration with the Gran Sasso Science Institute in L'Aquila, Italy.

This thesis is organized into two parts. Part I is an overview of the papers included in the part Part II of the thesis. Part I motivates our research, provides background about the modeling field and machine learning, presents our contributions, and discusses the state of the art and future work. Part II consists of a collection of published and peer-reviewed research papers included in this thesis.

Paper A A. Barriga, A. Rutle, and R. Heldal. Improving Model Repair through Experience Sharing. In Journal of Object Technology, Volume 19, Number 2, 2020.

Paper B L. Iovino, A. Barriga, A. Rutle, and R. Heldal. Model Repair with Quality-Based Reinforcement Learning. In Journal of Object Technology, Volume 19, Number 2, 2020.

Paper C A. Barriga, L. Mandow, J.L. Perez de la Cruz, A. Rutle, R. Heldal and L. Iovino. A comparative study of reinforcement learning techniques to repair models. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. 2020.

Paper D A. Barriga, R. Heldal, L. Iovino, M. Marthinsen and A. Rutle. An extensible framework for customizable model repair. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. 2020.

Paper E A. Barriga, L. Bettini, L. Iovino, A. Rutle and R. Heldal. Addressing the trade off between smells and quality when refactoring class diagrams. In Journal of Object Technology, Volume 20, Number 3, 2021.

ACKNOWLEDGMENTS

It is crazy to think that, after four years, two countries, three cities, six apartments, a surgery, and a worldwide pandemic, this journey is finally over. Reaching this point would have been impossible without the people that surrounded me these years, both in my professional and personal life, to whom I will be forever grateful.

First of all, I would like to thank my supervisors: Rogardt Heldal and Adrian Rutle. You know this thesis would not exist without your help and support. Both of you are fantastic professionals, but what has taught me the most during these years is how great you are on the inside. Rogardt, thank you for thinking as no one else does, for always believing in my ideas, and for valuing the weirdness in me from day one. Adrian, I honestly think you are one of the kindest persons I will know in my whole life, you have the gift to make others feel welcome and warm, even in the coldest of the countries. I will always feel happy in my heart remembering you both.

Apart from my supervisors, this thesis was possible thanks to Lawrence Mandow and José Luis Pérez de la Cruz. Thank you for welcoming me during my research stay at the Universidad de Málaga. I learned a lot from both of you. Thank you for being there for me even in the difficult times that were the first days of the pandemic in Spain. Also, thanks to Ludovico Iovino, for such a great and fruitful collaboration. If this thesis is a collection of papers is thanks to the ideas we built together.

I will never forget the support and inspiration I received from my former professors and colleagues at the Universidad de Extremadura: Elena Jurado, Juanma Murillo, Juan Hernández, and the teams at Quercus and SPILab. You offered me with my first opportunities as a researcher and encouraged me to always go beyond. I will always be grateful for everything you taught me. Thank you.

I also feel grateful for the support and help I received during these years from the staff at the Department of Computer Science, Electrical Engineering and Mathematical Sciences of the HVL. You have created a great working environment and you always made me feel comfortable. You were always available and helpful in those times where I needed your help. Thank you Kristin, Pål, Lars, Håvard, Violet and Volker.

These years were fun thanks to the people I met at the university, especially the rest of PhDs candidates. First of all, I cannot express with enough gratitude how great was meeting Fernando. You were (and still are!) the greatest senpai that shed light on the dark path that opened in front of me. Your friendship is one of the greatest things I got from these years.

To Frikk, thank you for being such a nice friend and helping out with everything we needed while we were out of Norway. It was super fun to have you around, and I hope we can keep this friendship and meet again in the future.

I also want to give my thanks to all the other colleagues from the office and the people I met thanks to them: Rui, Simon, Lucas, Iril, Anton, Håkon, Remco, Patrick, Larissa, Pedro, Salah, Suresh... thank you for the time we spent together, the laughing and the beers we shared.

This adventure would have been impossible without my parents, which always loved and supported me no matter what. Thanks to the education they fought to give

me I became the person I am today. Thank you for understanding that I had to leave, for bearing with the pain of having me far from home, and for always welcoming me when I came back. Knowing that someone is missing you unconditionally and is eager to see you again is priceless. I am so lucky for having you both. Thank you for being so patient with me, for being proud of me, for everything.

I also feel grateful for the friends who supported me from Spain all this time. If I still have some mental sanity is thanks to you. To my long-time school friends: Bea, Clara, Inma, Guada, María, Mer and Mónica. Keeping this group together after all these years is such a joy. Thank you for all the good times and the support you always give me. Thanks to Mer and Jorge for being the first visitors we had at Bergen, those days were fantastic.

To my Pokémon guys: José, Jorge, Kewy, Sergio and Piñazo. For the laughing and the great debates. What a treasure it is to find people which enjoy the same things as you do. To Papy and Noemí, for visiting us and for always being looking forward to meeting again. From now on it will be easier to meet.

Thank you, Joaquín. I have told you numerous times: your friendship is such a gift. Thank you for everything you have given me so far, I owe you a lot. You have no idea how much I cherish every afternoon spent by your side.

Finally, to Alex. For walking every step of this journey with me. Without you, this would have been impossible. I know that having you, I can overcome anything. It does not matter in which apartment, city, or country I am. If it is with you I feel at home. Thank you for not allowing me to give up, for your unconditional love, for the patience, for being family. Now, to our next adventure.

ABSTRACT

In model-driven software engineering, models are used in all phases of the development process. These models must hold a high quality since the implementation of the systems they represent relies on them. Models may get broken due to various editions throughout their life-cycle, and preserving their quality is crucial. Several existing tools reduce the burden of manually dealing with issues that affect models' quality, such as syntax errors, model smells, and inadequate structures. However, the same issues might not have the same solutions in all contexts due to different user preferences and business policies. **Personalization** would enhance the usability of **automatic** repairs in different contexts while reusing the experience from previous repairs would avoid duplicated calculations when facing similar issues. In addition, the variety of model types together with the variety of potential issues require model repair tools and approaches which are flexible, extendible, and customizable.

To this end, this thesis will focus on investigating the application of **machine learning** (ML) for repairing models. Our aim is to build a **model repair** approach that (i) provides automatic model repair, (ii) allows for user personalization, and (iii) may be extended to support the repair of different types of models which possess different kinds of issues. As a result, this thesis contains theoretical and practical contributions regarding the application of ML in model repair and the design of **a personalizable and extensible model repair framework**.

Applying ML in model repair is not a straightforward process, as most ML algorithms require a large amount of labeled data which is still unavailable in the modeling field. Hence, we propose the use of **reinforcement learning** (RL) algorithms, which can learn to solve a problem by directly interacting with it, without needing to train on labeled data. To improve the performance by reusing experience from previous repairs, we also implement a transfer learning (TL) approach.

In order to provide personalizable and extensible model repair, we have designed a modular framework: **PARMOREL**, where we integrate our RL and TL implementations. In PARMOREL, users can specify their repair settings before or after the repair and change the repair result by giving feedback to the framework. We extend and evaluate PARMOREL's modules through a series of implementations. The results show that we achieve automatic and personalized model repair and that PARMOREL supports different configurations of types of models, issues, actions, preferences, and learning algorithms.

SAMMENDRAG

I modelldrevet programvareutvikling brukes det modeller i alle faser av utviklingsprosessen. Disse modellene representerer programvare systemer, dermed er kvaliteten til disse systemene avhengige av kvaliteten til modellene. Det kan oppstå feil og mangler i disse modellene, ofte på grunn av uunngåelige endringer som må gjennomføres som del av utviklingsprosessen. Reparasjon av disse feilene er avgjørende for å bevare kvaliteten til systemene de representerer. Flere eksisterende verktøy reduserer den byrden som er knyttet til manuell håndtering og reparasjon av feil i modeller. Men automatisering alene er ikke tilstrekkelig siden de samme feilene kan ha forskjellige løsninger i ulike sammenhenger avhengig av brukerpreferanser. **Personalisering** forbedrer derfor brukervennligheten til **automatiske** reparasjonsprosesser, samtidig er det viktig å gjenbruke erfaringer fra tidligere reparasjoner for å unngå duplisering av beregninger spesielt når mann står overfor lignende feil. I tillegg er fleksible og utvidbare modellreparasjonsverktøy meget viktige for å løse feil i forskjellige modelltyper.

For å oppnå personalisering, automatisering, utvidbarhet og fleksibilitet i modellreparasjon, vil vi i denne avhandlingen anvende **maskinlæring** (ML). Vårt mål er å utvikle en tilnærming som (i) sørger for automatisk modellreparasjon, (ii) tillater brukertilpasning, og (iii) kan utvides for å støtte reparasjon av forskjellige typer modeller som innehar forskjellige typer feil.

Denne avhandlingen inneholder teoretiske og praktiske bidrag vedrørende anvendelse av ML i modellreparasjon og utforming av et **personlig tilpasset og utvidbart modellreparasjonsrammeverk** som vi kaller **PARMOREL**—Personalized and Automatic Repair of Models using Reinforcement Learning.

Å bruke ML i modellreparasjon er ikke en enkel oppgave ettersom de fleste ML-algoritmer krever en stor mengde data, noe som emmå ikke er tilgjengelig i modelleringsfagfeltet. Derfor har vi valgt å bruke **forsterket læring** (RL— Reinforcement Learning) algoritmer, som kan lære å løse problemer uten å trene på merkede data. For å bruke RL til å reparere modeller, formaliserer vi først modellreparasjonsproblemet som en Markov beslutningsprosess (MDP—Markov Decision Process). Deretter definerer vi teorien for å løse MDP ved hjelp av RL. For gjenbruk av erfaringer fra tidligere reparasjoner, har vi også implementert en tilnærming for å overføre læring (TL—Transfer Learning).

I vårt rammeverk PARMOREL, hvor vi integrerer og anvender RL og TL, kan brukerne spesifisere sine reparasjonsinnstillinger før eller etter reparasjonen, og endre reparasjonsresultatet ved å gi tilbakemeldinger til rammeverket. Vi evaluerer PARMORELs moduler gjennom en rekke utvidelser og eksperimenter. Resultatene viser at vi oppnår automatisk og person tilpasset reparasjon av modeller, og at PARMOREL støtter forskjellige konfigurasjoner av modelltyper, feiltyper, handlinger, preferanser og læringsalgoritmer.

Contents

Part I

OVERVIEW

Don't do anything that's been done before.
—Hiroshi Yamauchi

CHAPTER 1

INTRODUCTION

Since the end of the 20th century, society has entered an era where computing has gradually become an essential part of daily life, to the point where the digital merges with the fabric of the real world. As computation adoption grows, so does its complexity. To manage this ever-increasing complexity, computing experts have created different mechanisms to ease the development of software systems and to assure the correctness in their design and operation. These mechanisms often focus on abstracting software development from low-level code. By adding abstraction layers, software development has evolved from binary code instructions to software models. In addition to abstraction, these models may be used to automatically generate (parts) of the software, to analyse the software's properties before implementation, and for communication purposes.

When designing models, developers may introduce various issues which corrupt the models and reduce their overall quality. The chances of corrupting a model increase along with the size of the development teams and the number of software requirements. Also, the lack of coordination, misunderstanding, and mishandled collaborative projects can lead to the corruption of models [90]. These issues may lead to severe challenges in the later phases of the development process. Hence, solving these issues is essential in modeling, which is the focus of the model repair field.

One of the main challenges in model repair is that, for any given set of issues, there (possibly) exists an overwhelming number of repair actions that resolve them [62]. Hence, finding the appropriate actions to repair a model is not a trivial process.

As a consequence, a variety of approaches to model repair has been proposed over the last decades [61, 72, 75]. Most of the approaches focus on a specific type of issues existing in a specific kind of models. Additionally, as stated before, issues might have multiple repair solutions that do not satisfy all modelers' preferences. This situation has been tackled from two perspectives: on the one hand, the modeling community has developed a series of metrics and characteristics that can be used to get a measure of the quality of the solution, such as analyzability, adaptability, and understandability [20, 59], model distance [4, 89] or coupling [94]; on the other hand, the approaches in the literature offer some degree of personalization in the repair process [32, 61, 75].

Depending on the degree of personalization offered, some approaches work as support systems where the repair choice is fully left to the user's criteria [75], automatic approaches with some degree of interaction (before, during, or after the repair) [32], and fully automatic, non-interactive model repair [61]. Each approach presents advantages and disadvantages. Support systems that personalize the repair process provide

tailor-made solutions, however, they are time-consuming since they require close interaction from the modeler and are hard to scale for repairing a wider range of models. Automatic solutions improve repair time, however, they have the drawback of providing the same solutions for the same errors although different modelers may have different preferences for repairing the same model. A desirable solution should provide a balance between automation and personalization of repair [62], facilitating the use of both approaches' advantages.

In this direction, cognifying model repair [30] could be a solution to establish a balance between automation and personalization. Cabot et. al define cognification as *"the application of knowledge to boost the performance and impact of a process, (...) including the combination of past and current human intelligence"*. Currently, cognification is mostly achieved by applying machine learning (ML) algorithms.

Over the last years, ML has risen as a disruptive trend that is bringing a new era for how we design and maintain technology. ML brings the potential to automate complex tasks and replicate human behaviour in more and more domains, including software engineering processes. ML's ability to recognize patterns and learn how to deal with them allows automating repetitive tasks such as bug fixing and code repair [85], testing [34], quality assurance [33, 63], or verification [44]. Despite ML's potential to provide automatic model repair, there is not much research done in this direction.

Applying ML within modeling has already been identified as a challenge for the community by authors in [28, 30, 70]. The main challenge for ML adoption in modeling in general, not only in model repair, is about data availability [29, 45]. Most well-known ML algorithms depend on large amounts of data to learn how to repair a problem [66]. Due to this lack of data, the type of ML algorithms feasible to apply to model repair is reduced to those that do not require large amounts or labeled data to solve a problem.

In this thesis, the research idea is to study how reinforcement learning (RL), an ML branch that does not require labeled data to solve a problem, can be applied to enhance model repair. In order to deal with the different possible repair solutions that may exist for an issue, we develop an automatic approach, using RL, that allows users to influence the repair results—personalizing them to their requirements. Furthermore, we design a framework with enough flexibility to handle the above-mentioned variations in the model repair problem; i.e., types of models, issues, repair preferences, and actions.

1.1 Research questions

In this thesis, we focus on applying RL to model repair and developing an approach that achieves personalized and extensible model repair. Specifically, we focus on the following research questions (RQs):

- **RQ1:** How can RL algorithms improve model repair? How can we apply RL algorithms in model repair?
- **RQ2:** How can we keep a human in the loop when performing model repair? How well can human intervention improve results?
- **RQ3:** How can a model repair framework be designed to tackle the variety of repair situations existing in the model repair field?

RQ1 is concerned with the investigation of RL algorithms, how they can be applied into model repair and how useful they can be in solving current model repair challenges. RL offers a series of features that could solve model repair challenges, such as the ability to solve a wide range of problems without following a tailor-made approach, offering equilibrium between automation and a manual approach, and abstracting how modelers can deal with the model repair process. Therefore, the goal of our research in **RQ1** is to identify current challenges in the model repair field that can be solved by using RL, to study which RL algorithms these challenges can be addressed with, and to build the theoretical foundation that allows to apply the identified algorithms into the model repair problem.

RQ2 aims to explore and identify which mechanisms are most beneficial to keep a human in the loop while repairing models and how this interaction affects the final results. We will show how that can be done with the help of RL, creating an equilibrium between automation and personalization. To this end, we will allow users to introduce their repair preferences so that the system can learn how to automatically repair models following those preferences. Hence, the goal of **RQ2** is to research how human knowledge can be incorporated into the RL algorithm when repairing.

RQ3 focuses on researching how the outcomes of **RQ1** and **RQ2** can be incorporated into a model repair framework. The existence of tailor-made solutions has already been identified as a challenge in the modeling field by the authors in [27]. Within model repair, implementing tailor-made tools to deal with the wide range of existing elements in the field is time and resources consuming. Hence, the goal of **RQ3** is the implementation of a unifying framework for model repair that addresses the problems of tailor-made approaches.

1.2 Outline

This thesis is organized into two parts. Part I motivates our research, provides background about modeling, model repair, ML, presents our contributions, and discusses the state of the art and future work. The outline of Part I is organized as follows:

Chapter 2: Modeling background

This chapter introduces the necessary background for understanding the terminology and main concepts used throughout the thesis. We start with a brief explanation of models and model types. Then, we continue by presenting different model issues and model measurement techniques.

Chapter 3: Machine learning background

This chapter presents background of ML, presenting different algorithms and ML-related concepts. Specifically, we focus on explaining RL, Markov decision process (MDP), and transfer learning (TL).

Chapter 4: Solving the model repair problem

This chapter describes our theoretical contribution, presenting how we have adapted MDP, RL, and TL concepts to solve the model repair problem. Furthermore, we introduce part of our practical contribution by explaining how we have implemented RL and TL in our approach.

Chapter 5: The PARMOREL framework

This chapter presents the remaining part of our practical contribution: PARMOREL, an extensible and personalizable model repair framework powered by RL. Here, we motivate and discuss the modular architecture of the framework and the extensions of each of its modules.

Chapter 6: Research method

This chapter discusses the research methodology we have followed for developing this thesis. We also discuss our evaluation plan and present how we have selected the data used in our experiments.

Chapter 7: Contributions

This chapter presents the contributions of our thesis. We start with a summary of the papers written and the experiments conducted during the development of this thesis. Then, we present an overview and discussion of the results obtained. Lastly, we introduce the limitations and threats to the validity of our work.

Chapter 8: Related work

In this chapter, we discuss related work. We classify existing model repair approaches by the technique they use to perform repair, distinguishing between non-ML and ML approaches. We put our work in context by relating each presented approach with PARMOREL.

Chapter 9: Conclusions and future work

This chapter re-visits our RQs and provides a summary of the main contributions of this thesis. We also outline several future research directions based on the work undertaken for this thesis and current ML challenges.

Part II consists of a collection of papers produced during the development of this thesis: five published and peer-reviewed articles [9, 11, 13, 18, 49] (papers A - E). The outline of Part II is organized as follows:

Paper A: Improving model repair through experience sharing

Paper B: Model repair with quality-based reinforcement learning

Paper C: A comparative study of reinforcement learning techniques to repair models

Paper D: An extensible framework for customizable model repair

Paper E: Addressing the trade off between smells and quality when refactoring class diagrams

1.3 Suplementary material

In addition to the articles included in the Part II of this thesis, we have published three workshop articles and a short paper presenting initial and alternative research directions, which results overlap with later papers. Furthermore, we have submitted two articles to an international journal, which are in the second round of the reviewing process at the moment of writing this thesis.

1. The paper [14], presented at the Workshop on Analytics and Mining of Model Repositories (AMMoRe) in October of 2018. This paper presents an early prototype of how ML could be applied to model repair, reflecting on its benefit on repair time.

2. The paper [15], presented at the Workshop on Artificial Intelligence and Model-driven Engineering (MDEIntelligence) in September of 2019. This paper shows an early prototype of PARMOREL where it is applied to repair conformance issues in a group of mutant models.

3. The paper [16], presented at the Jornadas de Ingeniería del Software y Bases de Datos (JISBD) in September of 2019, is a short paper that presents the idea to produce repaired models of higher quality than the original ones.

4. The paper [10] was accepted at the 14th Workshop on Models and Evolution (ME), co-located at the Models Conference in October 2020. In this paper, we present a dataset of 2420 models, which has been collected for experimenting with different approaches conceived by the authors, mostly coming from GitHub and the ATL zoo [5]. Additionally, we present a tool-chain to analyze datasets of models, obtaining which issues (if any) are present in each model and measuring quality characteristics. Some of the models in the dataset and the measurement of quality characteristics are used in later papers.

5. The paper [12] was submitted to the International Journal on Software and Systems Modeling (SoSyM) in March 2021, as an extension of Paper D and including the results of Paper E. This paper goes beyond these previous papers, exploring PARMOREL's full extensible potential, detailing its modules and submodules, and including new extensions that we detail in the Appendix.

6. The paper [17] was submitted to the SoSyM journal in April 2021. In this paper, we explore the state of the art in the intersection between artificial intelligence (AI) and model repair, the AI-powered model repair field. From existing literature in this field, we identify a series of challenges which the modeling community needs to overcome to extend the adoption of AI in the model repair field.

The papers [10, 12, 14–17] are not formally part of this thesis and will hence not be discussed further. The models used as dataset and the implementation of PARMOREL can be found on our project's website [2].

CHAPTER 2

MODELING BACKGROUND

In this chapter, we provide background information necessary to understand the modeling field. We focus on those modeling aspects relevant to this thesis, explaining concepts such as models and details of some types of models (class and sequence diagrams), different kinds of issues that might appear in models (conformance issues, smells, and inter-model inconsistencies), and a couple of model measurement techniques (quality characteristics and model distance).

2.1 Models

Models allow to represent a part of reality in a simplified way, making them useful artefacts for documentation, communication, and supporting the development of a project. In software engineering, models are used to tackle the complexity of software systems by abstracting and hiding their underlying technology as much as possible.

There exist different types of models based on their purposes and modeling languages, (e.g. Ecore [88], the Unified Modeling Language (UML) [82], the Business Process Model and Notation (BPMN) [37], etc). In this thesis, we focus on Ecore class diagrams and UML class and sequence diagrams. The terminology we use is taken from the Eclipse Modeling Framework (EMF) [88]. The purpose of this section is not to give the complete syntax and semantics of class and sequence diagrams, but to give enough information to understand our examples used later in the thesis.

2.1.1 *Class diagrams*

Class diagrams show the static structure of a system, the elements that exist (classes), and their relationships (references) [1]. We use as an example the class diagram in Fig. 2.1, representing students, books, and how they relate to each other.

- Class: A class represents a concept within the system being modeled. Classes have attributes, operations, and references to other classes. A class is typically represented as a box with a name. Two classes appear in Fig. 2.1: *Student* and *Book*.
  - Attribute: A class-associated property used to hold knowledge about the class. To be correctly defined, attributes must have a name and a data type

(e.g., String, Integer, etc). In Fig. 2.1, within the class *Student*, there are two attributes: *name* and *age* which types are String and Integer, respectively.

- Operation: An operation represents a functionality supported by the class. They work as placeholders for methods in the object-oriented paradigm. In Fig. 2.1, within the class *Student*, there is the operation *readBook()*.

- Reference: A reference represents a relationship between two classes, where one of the classes is the source and the other one is the target of the reference. The multiplicity of a reference is a constraint indicating how many objects may participate in the relationship. In Fig. 2.1, the classes are related with two references which establish that a book is *owned by* exactly 1 student and that each student *owns* at least one book.

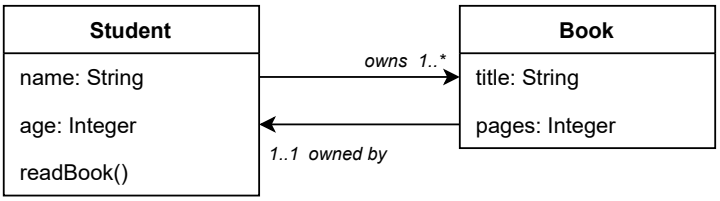


Fig. 2.1: Sample class diagram representing students, books and their relationship

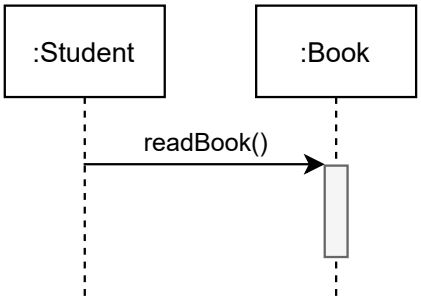


Fig. 2.2: Sample sequence diagram representing how a student interacts with a book

2.1.2 *Sequence diagrams*

Sequence diagrams represent interactions in a system [1]. The main elements in a sequence diagram are lifelines and messages. We use as an example the sequence diagram in Fig. 2.2, representing how a student interacts with a book.

- Lifeline: A lifeline represents an object which participates in an interaction. Classes in class diagrams have their corresponding lifelines in sequence diagrams. In Fig. 2.2 we represent two lifelines: *Student* and *Book*.

- Message: A message defines a communication between the lifelines of an interaction. A message represents an invocation of an operation on a target lifeline (see *readBook()* in Fig. 2.2).

2.2 Model issues

In this thesis, we refer to anything that might be considered as wrong or improvable within a model as *issue*. Issues might be of different types depending on their nature and the context where they appear. In this section, we focus on the types of issues relevant to this thesis.

2.2.1 Conformance issues

Models are defined using modeling languages, which may be represented by metamodels. A metamodel is again a model that specifies the concepts and the relationships of the modeling language. The metamodel defines constraints that any model created from it needs to conform to for being valid. In addition, the metamodel often contains extra constraints in a formal language that models also need to conform to. Thus, there is a conformance issue when a model violates one or more constraints imposed by the metamodel and the formal language.

In this thesis, we focus on conformance issues with respect to the Ecore metamodel [88]. An example of these issues is when an attribute does not have a data type, violating the *typed attribute* constraint of the Ecore metamodel.

2.2.2 Smells

Smells in code [22] are not bugs or errors but instead, can be considered as violations of the fundamentals of developing software that decrease the quality of code. In a similar way, smells in models [24] are indicators that there might be issues within the model design. Examples of smells in models would be classes that are isolated from the rest of the model and redundant relationships between classes.

Smells may severely affect the maintenance and evolution of the models. Therefore, their early identification and removal are crucial to assure the final quality of the models. There are many smells defined in the literature [22, 68] and detecting and removing them is far from trivial, as their removal may have a negative impact on the overall model quality. In this thesis, we focus on selective removal of smells to avoid deteriorating the models' quality.

2.2.3 Inter-model consistency

Multiple models might be needed to represent software systems from different perspectives. In this case, models need to be consistent with respect to each other [42]. For example, one could use various UML models to describe different aspects of software: class diagrams to describe the structure and sequence diagrams to define (parts of) the interaction between the elements of the software (e.g., see figures 2.1 and 2.2). An issue might arise if, for example, the operations in the class diagram do not match the messages in the sequence diagram [92]. Since both diagrams describe the same system, they should be kept consistent with each other. In this thesis, we consider inter-model consistency between UML class and sequence diagrams [92].

2.3 Model measurement

Finding the best solution for a given issue in a model is not a trivial task since there might be multiple repair solutions that a modeler could choose, while there might not exist an objectively best solution to satisfy all modelers.

However, the modeling community has developed a series of measurement techniques that can be used to get a measure of a model's quality. This measurement can be used to determine which repair option might be more satisfactory for modelers with interests in specific aspects of the model. In this section, we present two measurement techniques relevant to this thesis.

2.3.1 *Quality characteristics*

In [19, 20, 59, 94] different works propose quality characteristics specifically conceived to measure the quality of models and other modeling artifacts. In these works, characteristics like maintainability, usability, coupling, and cohesion are introduced. Some of these characteristics measure the general quality of the model, such as maintainability, and others measure specific parts of them, such as the coupling of a class, measuring the degree of interdependence between the classes of a model.

Multiple quality characteristics may be combined and measured in order to evaluate qualitative aspects of the models. These aspects are particularly relevant in the activity of model repair since the actions chosen to repair issues could produce valid models, but with low-quality characteristics. By using these characteristics, it would be possible to choose repair actions that boost quality aspects chosen by the modelers. In this thesis, we focus on the following quality characteristics: maintainability, reusability, understandability, complexity, and coupling.

2.3.2 *Model distance*

Model distance is a measurement technique that counts the differences in the structure of several models. The more similar the models, the closer the distance between them. Within model repair, model distance can be considered to preserve the original model structure while repairing, measuring the distance between the original model and its repaired versions. Actions that make the least changes in the model can be chosen following this technique, hence prioritizing a conservative repair. With this technique, undesired side-effects of the repair actions can be minimized, which can help in maintaining the model's quality [52, 58, 90].

The task of measuring the distance between two or more models can be managed by specific distance metrics, inspired by distances between words and graphs [43] or by matching algorithms [57]. In this thesis, we make use of two model distance implementations based on matching algorithms, inspired by the Levenshtein edit distance formula [57].

CHAPTER 3

MACHINE LEARNING BACKGROUND

In this chapter, we provide background on ML, discussing different algorithms classified by the way they learn: supervised learning, unsupervised learning, and RL. This is followed by a more detailed discussion about RL concepts and algorithms since they are the focus of this thesis. Furthermore, we discuss other ML concepts that are relevant in this thesis, such as MDP and TL.

ML is a subfield of AI that allows machines to learn from data without being programmed explicitly, achieving automated detection of meaningful patterns in data. ML focuses on the use of the strengths and special abilities of computers to complement human intelligence, often performing tasks that fall way beyond human capabilities (e.g., processing a great amount of information in a tiny timespan).

There exist multiple ML algorithms, which can be classified depending on how they learn from data [86]. Figure 3.1 summarizes this classification, including the data requirements for each type of algorithm and their usefulness. In the following, we briefly explain these three groups.

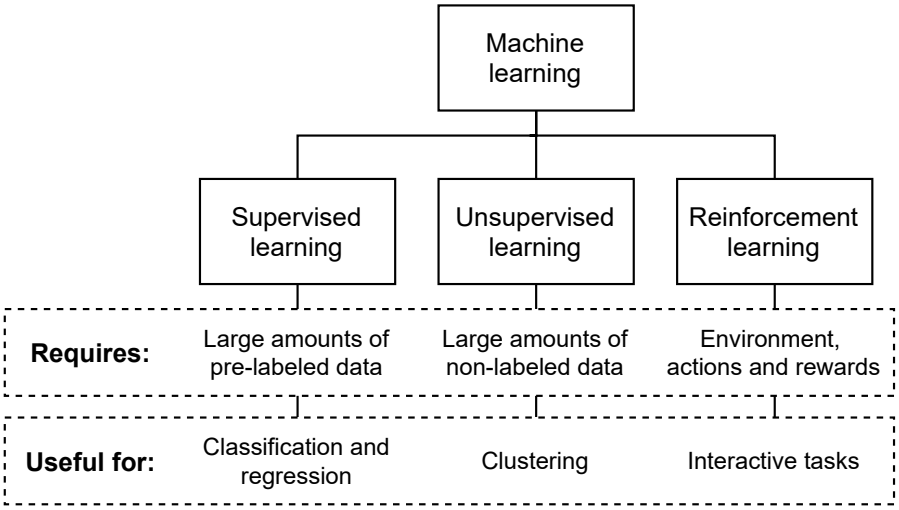


Fig. 3.1: ML algorithms classified by learning type, including their data requirements and usefulness

3.1 Supervised learning

Supervised learning algorithms learn a function that maps an input to an output based on example input-output pairs. They infer a function from pre-labeled training data consisting of a set of training examples [66]. Supervised learning algorithms usually require large amounts of pre-labeled data in order to learn how to solve a problem. This sort of algorithms is 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 [31]. An example of a classification task would be distinguishing pictures of cats and dogs. Even though all cats and dogs 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, distinguishing features, etc, and a label specifying the correct prediction (e.g., whether the picture contains a cat or a dog). A successful algorithm should be able to recognize previously unseen cats and dogs after training on this data.

In regression (also called prediction), the algorithm takes an input and returns a number as output [6]. An example of a regression task would be to predict the price of housing in ten years. The algorithm could train on data containing attributes such as the size of the houses, initial value, evolution during the years, quality of the materials, etc. Then, the algorithm would fit a function to this data to predict the output price.

3.2 Unsupervised learning

Unsupervised learning is a type of algorithm that learns patterns from unlabeled data. These algorithms search for recurrent patterns in the input data [86]. Then, as output, unsupervised algorithms group the introduced data, displaying which data points in the input data are more related to each other. The more data, the better these algorithms work. The most common application of unsupervised learning is clustering.

Clustering is a task for finding clusters/groups within the input data [6]. An example application is image compression, where the objective is to reduce the file size of the image. The input is the pixels that make up the image, and each pixel is represented by an RGB value. The algorithm will find all the pixels related to a specific color (e.g., red), calculate the average color, and set it for all the pixels of that color. This will slightly reduce the details of the image, although in an undetectable way for the human eye, and save storage space.

3.3 Reinforcement learning

As stated in Chapter 1, the lack of modeling data makes it difficult to apply most ML algorithms, especially those within supervised and unsupervised learning. Hence, we focus and provide more information about RL, an ML branch of algorithms that do not require data to solve a problem.

RL consists of algorithms able to learn by themselves how to interact in an environment without existing pre-labeled data, only needing a set of available actions and

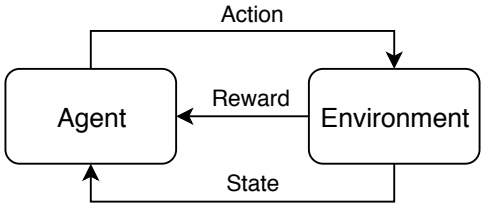


Fig. 3.2: Agent-environment interaction in RL, © 2019 IEEE

rewards for each of these actions [91]. By using rewards, these algorithms can learn which are the best actions to interact with an environment. The learning process of RL comes from the interaction between an agent (the intelligence in the algorithm) and an environment (the problem to solve). Figure 3.2 displays this interaction.

For example, if the *environment* is a road, the *agent* is a self-driving car, the *action* is turning one meter to the right and the *state* the current position of the car in the road, then, the new state would be the car's new location: one meter to the right with respect to the previous position. If the action is positive for the car (avoiding an obstacle), it receives a *reward*, contrarily (stepping on a wall) it is penalized. The agent will continue performing actions trying to get the highest reward until it reaches its ultimate goal; e.g., arriving to the car's destination.

In the following, we present some concepts important to understanding RL:

1. State space: Set of states, observable situations, that can happen in a system. Every system has an initial state (how it starts) and a final state. It is important to differentiate between a state and the actual system. A state is what is observable by the agent and it might not contain all details about the system, because they might not be necessary or available to the agent.

2. Action space: The set of actions that can modify the system, leading to new states.

3. Reward: A numerical value that tells the agent how good is the action it applies.

4. Step: A step corresponds to the application of one action in the system.

5. Episode: Each episode corresponds to one iteration in which the algorithm has successfully reached the final state using the available actions. Hence, an episode ends when the final state is reached. The number of episodes is finite; the algorithm is provided with a maximum number of episodes to run. A good number of episodes is when the algorithm has sufficient time to find the optimal sequence of actions to reach the final state. It is not straightforward to conclude what number of episodes is the right one in a given context [91] and hence, it needs to be defined empirically through experimentation.

RL is a wide field with many different algorithms, in this thesis, we focus on tabular algorithms, such as Q-Learning and Q(λ) [91]. These algorithms are called "tabular" because they store the knowledge acquired about how to solve a problem in a table structure. This structure is referred to as Q-Table. The Q-table stores pairs of states and actions together with a Q-value. The Q-value is calculated using the rewards and it indicates how good each pair is, that is, how good an action is for a given state. More details about these algorithms can be found in Paper C.

3.4 Markov decision process

MDPs are mathematical models used to solve sequential decision-making problems [69]. At specified points in time, a decision agent observes the state of a system and chooses an action. The action choice and the state make the system transition to a new state at a subsequent discrete point in time. The agent receives a reward signal at each transition. The goal of the MDP is to find a policy, or sequence of actions, that maximizes the rewards accumulated over time. The MDP concepts of state, action, and reward are abstract and must be defined in order to apply an MDP to solve a specific problem. RL algorithms such as Q-Learning and Q(λ) are used to solve MDP problems.

MDPs are defined in terms of a finite set of states and a finite set of actions. State transitions must depend only on the current state and the action chosen. The MDP is a theoretical framework and its concepts can be used to solve different problems. Each problem might require a different definition of the MDP concepts to be solved (e.g., a state could be a position in a maze, the score in a videogame, etc). Additionally, the same problem could be solved with different definitions of the same MDP concepts. For example, in the self-driving car problem, so far we have defined the state space as each position of the car in the road, but this is not the only state definition that could solve the problem, the state could for example be the history of the trajectory of the car.

3.5 Transfer learning

TL is a research line in ML that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem to solve it faster. TL allows reusing experience when the rewards change from one scenario to another [77]. For example, in the self-driving car problem, there might be roads with different obstacles to be avoided or with traffic signals in different locations.

TL differs from traditional ML in the fact that, instead of learning how to solve a problem from zero, it reuses experience gained in solving a source task (a known problem) to accelerate the solution of a new target task (an unknown problem). The benefits of TL are that it can speed up the time it takes to develop and train an ML system by reusing already developed solutions.

There exist many techniques within TL. The most common ones are starting-point and imitation methods [93]. Starting-point methods use the solution found in the source task to set the initial experience in a target task. Imitation methods use parts of the source task experience to influence the solution of the target task. Applied to Q-Learning and Q(λ), following starting-point methods the whole Q-table from a previous problem would be reused in a new one while following imitation methods only some parts of the source Q-table would be copied to the new problem.

CHAPTER 4

SOLVING THE MODEL REPAIR PROBLEM

In this chapter, we present how we have formalized the model repair problem as an MDP. Then, we present how we have adapted and implemented RL and TL concepts to model repair. The contents of this chapter are further detailed in papers A and C.

4.1 The model repair problem as a Markov decision process

In this section, we formalize the model repair problem as an MDP. As stated in Section 3.4, MDPs are mathematical models used to solve sequential decision-making problems. In the following, we revisit the concepts introduced in Section 3.3, redefining them to solve the model repair problem:

1. State space: The state space is defined by the set of issues present in the model. The initial state corresponds with the issues present in the model when the repair starts. Each state is hence defined by a set of model issues. This set is updated after each step with the current issues present in the model. The final state has an empty set of errors, i.e, it stands for a repaired model. An example of a state would be: {*The typed element X must have a type, There may not be two classifiers named X, A class may not be a superclass of itself*}.

2. Action space: The set of editing actions able to repair a model. Actions might come from an external tool, be defined by users, or be obtained from a modeling framework such as EMF [88], as we will see in the next chapter. For each state, actions are filtered, so that only actions capable of repairing at least one issue in the state are considered. Some examples of these actions, when dealing with conformance issues, are *delete, setName, setType, setContainment*, etc.

3. Reward: In every non-final step (not the last step in an episode) the reward will be 0. When the final state is reached, at the end of an episode, the reward will be given by an external tool used to measure some characteristics of the provisional repaired model generated in that episode. Rewards can be adapted to align with user preferences to personalize the repair result. Since rewards indicate how good actions are, the only requirement for user preferences is that they can be quantified (e.g., preserve the original model structure by minimizing the model distance metric or boost quality characteristics by optimising quality

characteristics). Users can choose their preferences before the repair process starts.

4. Step: A step corresponds with the application of one action to solve an issue or set of issues in the model.

5. Episode: An episode corresponds with an iteration in which the RL algorithm has successfully repaired the model, leaving no issues unsolved. Hence, an episode ends when the final state is reached.

4.2 Reinforcement learning applied to model repair

Once we have a formalization of the model repair problem as an MDP, we need to adapt and implement the concepts of RL to solve the problem. In this section, we present our implementation of how to apply RL to repair models. We have used tabular RL algorithms, such as Q-Learning and Q(λ) [91]. As we saw in Chapter 3, tabular algorithms store the knowledge acquired about how to solve a problem in a table structure, the Q-table. This table stores pairs of states and actions together with a Q-value [((State, Action), Q-value)]. The Q-value is calculated using the rewards, and it indicates how good a pair is, that is, how good an action is for a state where the action is applied.

Now, we will go through how the model repair process works step by step with the help of Fig. 4.1. As we can see, the repair process receives as input the input model to repair and user preferences. Then, the repair process starts with the action Extract issues, which extracts issues from the input model. Following the arrow *filter*, the action Obtain actions obtains available editing actions. For each of these actions, following the arrow *actions*, Check Q-table checks whether a pair with the current state and action exists already in the Q-table. If it does not exist, following the arrow *if pair (state-action) does not exist*, Add to Q-table is triggered, adding the pair to the Q-table. This way, the Q-table will contain a pair for every available editing action and the current state of the model.

Next, following the arrow *Q-table* or if the pair already existed in the Q-table, following the arrow *if pair (state-action) exists*, Select action is triggered and one of the actions stored in the pairs of the Q-table is selected to be applied in the model. Since we follow an ϵ-greedy strategy, actions will be selected either by having the highest Q-value or randomly in 30% of the cases (value of ϵ of 0.3, this value provided the best results according to our experiments in Paper C). This combination of exploitation and exploration allows to pick repair actions that otherwise might have never been selected.

Then, following the arrows *action*, the actions Apply action in model and Store action in the i^{th} episode sequence are triggered, applying the selected action in the model and storing it in a sequence of actions belonging to the current episode i. After applying the action, if there still are issues in the model, following the arrow *if issues left, reward* = 0, the action Update Q-table is triggered, updating the Q-table with a reward of 0 for the pair of the current state and selected action. Then, following the arrow *if issues left, use as input*, the algorithm starts again, receiving as input the model with the actions applied so far. This process of applying an action to address a set of issues constitutes a Step.

However, if there are no issues left in the model, following the arrow *if no issues left*, the action Obtain reward is triggered and, following the arrow *reward*, the pair will be updated in the Q-table receiving a reward according to the preferences chosen by the user. Also following the arrow *if no issues left*, the action Create i^{th} episode model creates a repaired version of the model with all the actions applied during the episode. Following the arrows *i^{th} episode model* and *i^{th} episode sequence*, the action Store as results is triggered, storing the repaired model and the sequence of selected actions obtained during the current Episode (the steps performed so far until no more issues are left in the model). With this, an episode finishes and a new one starts.

With the start of a new episode, the input model with its original issues is recovered and the repair process starts again, repeating the process inside the box Repair process, aiming to find new sequences of repair actions and getting further testing on the ones already found. The more an action is applied, the more trustiness will have its Q-value, as it will receive more rewards and hence the Q-value will be more accurate of how good that action was for the state it was applied to. To avoid reaching the maximum number of episodes needlessly (as security, we use between 1000 and 5000 episodes as maximum), we run the process with an early-stopping criteria. The process will stop once the maximum Q-value of the pairs including the initial state remains unchanged for 25 episodes (this value provided the best results according to our experiments in Paper C).

When all episodes finish, or the early-stopping criteria has been activated, following the arrow *input*, the action Select best result is triggered and the repair sequence with the highest Q-values will be selected. Lastly, following the arrow *output*, the corresponding provisional repaired model is saved as the Repaired model.

Additionally, for those situations where automatic repair or selecting preferences prior to the repair might not be enough for the users, they can manually select which sequence of actions they prefer among the repair sequences found in the episodes, following the arrow *optional feedback*. By doing this, extra rewards will be provided to the selected actions. This way, users can correct and influence how the RL algorithm behind the model repair process learns.

4.3 Transfer learning in model repair

In model repair, the value of each pair in the Q-table may depend on multiple rewards since it might involve several user preferences, e.g., a user might want to boost both the maintainability and reusability of a model. Introducing user preferences complicates reusing the experience acquired by the RL algorithm since what is a good repair for one user might not be acceptable for another one.

Working with the same Q-table in different repair scenarios is useful as long as user preferences remain unchanged. However, it is not convenient to directly reuse the Q-table (as in starting-point methods, see Section 3.5) when choosing new preferences, since the repair process would use the Q-values calculated with the old preferences and this could lead to repair decisions unaligned with the new ones. Following imitation methods would not be convenient either since we would still copy some of the Q-values from an old Q-table calculated with old preferences.

To overcome the limitations of both methods, we apply our own version of the

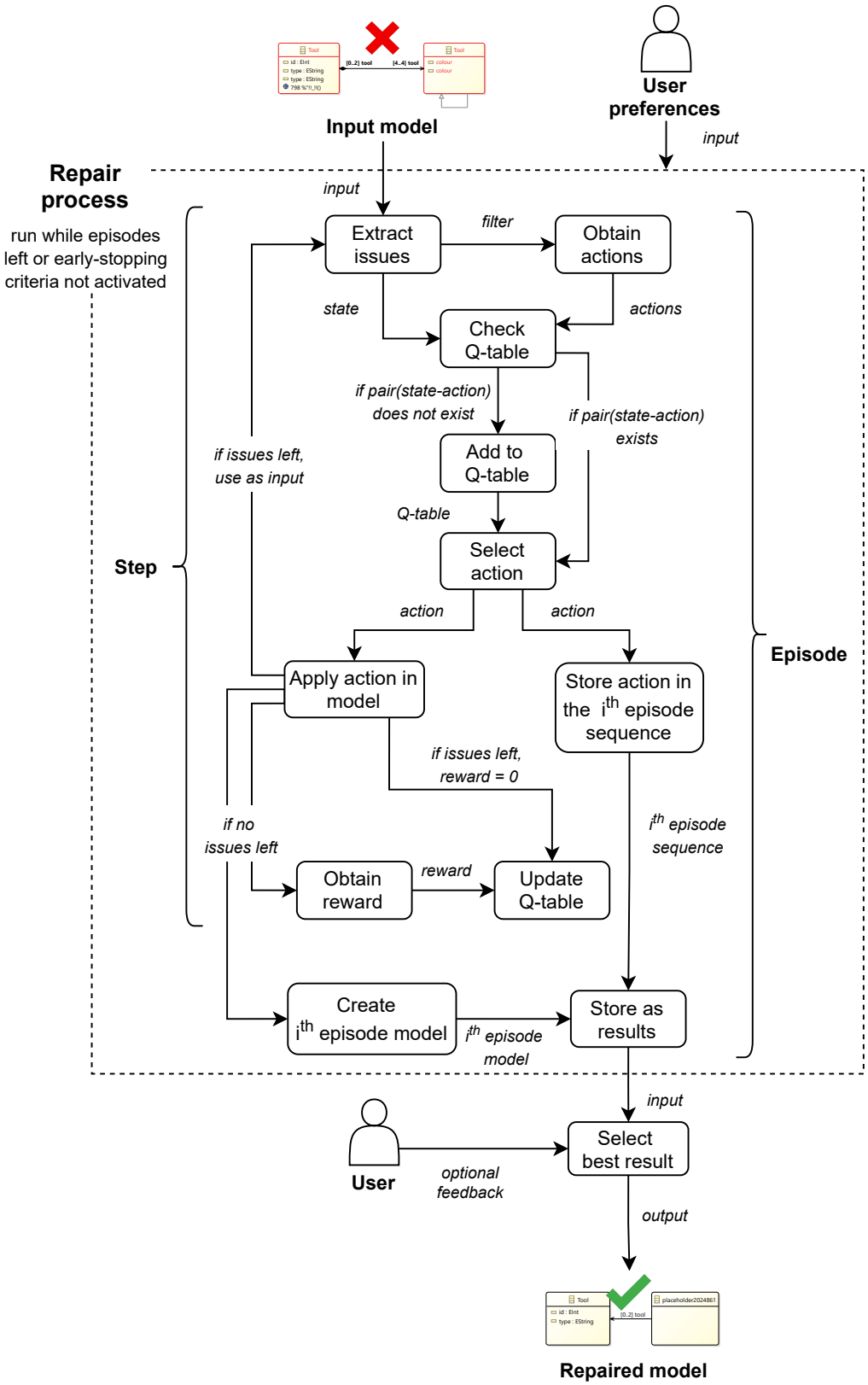


Fig. 4.1: Summary of the repair process using RL

starting-point method by copying all Q-table pairs of state-action without their Q-values, so that the algorithm would not start with an empty Q-table. In addition, we apply a variant of the imitation method in which instead of copying the Q-values from the Q-table, we keep track of which preferences were used to produce the Q-values, accumulate their values, and reuse those which are aligned with the new user preferences.

An example of applying TL to model repair is displayed in Fig. 4.2. In the left part of the image, we show the Q-table of *User1* once she finishes the repair process. *User1* chooses as preferences *pref1* and *pref2* to repair a model that contains two issues, namely *issue1* and *issue2*. Both issues can be repaired with actions *action1* and *action2*. Then, in the right part of Fig. 4.2 we show how the Q-table will look for *User2* once she starts repairing. This user chooses to repair with preferences *pref1* and *pref3*. The model to repair is different from the one repaired by *User1*, but since what is relevant for PARMOREL are issues and actions, the *Experience* can be reused regardless of the specific model to repair. Without TL, the Q-table will not exist and a new one will be created, adding more time to the processing part of the learning algorithm. With TL, every entry existent in the *Experience* is copied in the Q-table, and since *pref1* is shared with *User1*, the Q-table is initialized with the rewards provided from this preference multiplied by a discount factor (0.2 in Fig. 4.2). By using a discount factor, we assure previous repair processes influence the new repairs by jump-starting the repair process but do not interfere with learning new repair sequences.

As a consequence, by using TL, when the repair process starts for *User2*, the time spent in populating the Q-table is reduced and the learning algorithm will already have an intuition of which actions are better for each issue. More details about how TL works can be found in Paper A.

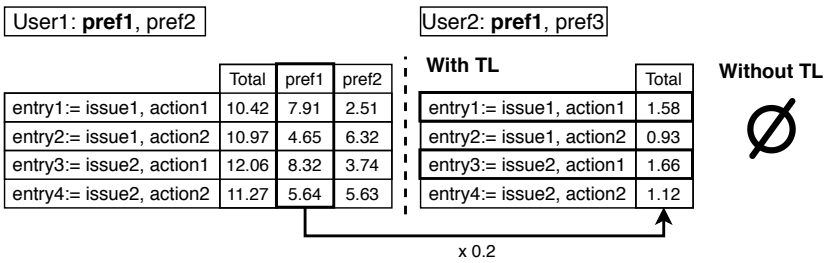


Fig. 4.2: Transfer learning between 2 users with a shared preference

CHAPTER 5

THE PARMOREL FRAMEWORK

In this chapter, we present our unified approach for model repair, the PARMOREL framework. We integrate within PARMOREL the RL and TL implementations presented in the previous chapter. PARMOREL stands for **P**ersonalized and **A**utomatic **R**epair of software **MO**dels using **RE**inforcement **L**earning. PARMOREL revolves around two core ideas: personalization of results and extensibility.

PARMOREL can be used by modelers to repair models in everyday modeling scenarios. Modelers can input their models in PARMOREL as they get broken. As PARMOREL is powered by RL, no training data is needed and the repair can be performed for a single model, without needing further input, or to as many models as the user wants at once. Since the framework is extensible, modelers can adapt it to their specific needs, repairing different types of models and issues. Moreover, modelers can adapt the repair to align with their requisites. If the modeler deals with a changing modeling environment, she can continue working with PARMOREL by adapting it, without needing to change between several tools. In the following, we present the modular architecture of PARMOREL and its modules' extensions.

5.1 Architecture

PARMOREL's architecture has been designed around three main concepts: issues to be found in the models, actions to be applied in response to issues, and preferences that the user can specify to customize how to address issues (see Fig. 5.1). The framework contains an RL algorithm, following the concepts presented in Chapter 4, in charge of learning and deciding which is the best action—among a set of available actions—to address a set of issues, according to the user's preferences.

The PARMOREL framework permits the issues, actions, preferences, and learning

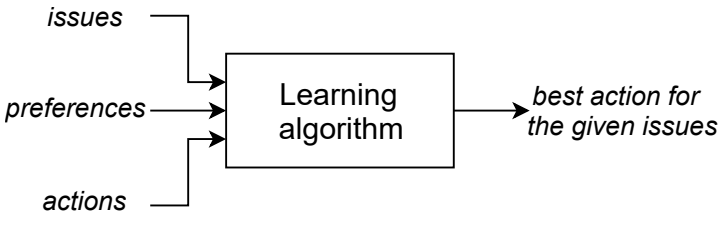


Fig. 5.1: Simplified workflow in PARMOREL

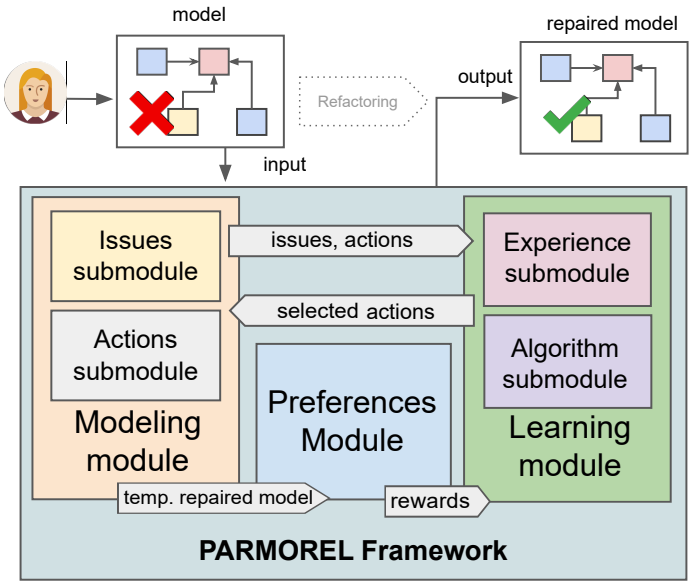


Fig. 5.2: PARMOREL's modular architecture

algorithm to be modified or changed based on the type of models to repair and the repair's goal. PARMOREL is implemented as an Eclipse plugin, following a modular architecture (see Fig. 5.2), and permits users to customize its modules through a series of interfaces. In this section, we explain each of the constituting modules of PARMOREL. PARMOREL's implementation included in this thesis can be found on our project's website [2].

5.1.1 *Modeling module*

The *modeling module* is in charge of validating and manipulating the models. This module is responsible for interacting with the models and providing information to the *learning module*. It sends information about actions available to modify the model and issues present in the model so that the *learning module* can learn what actions should address the issues. With the selected actions, the *modeling module* will create a temporary repaired model from which the *preference module* will extract rewards. The *modeling module* is divided into two submodules, namely the *issues submodule* and *actions submodule*, as showed in Fig. 5.2.

Issues submodule The *issues submodule* is in charge of identifying which issues are present in the model and send them to the *learning module* so that it can learn how to solve them. An issue could be a conformance issue, a smell, a violation with respect to an architectural pattern or a specific constraint, an inconsistency between two or more models, etc.

Actions submodule The *actions submodule* is in charge of sending to the *learning module* which actions are available for modifying the model. Additionally, it is also in charge of applying the actions chosen by the *learning module* to solve the issues identified by the *issues submodule*.

The *issues* and *actions submodules* are tightly connected, as actions are defined as an answer to the issues present in the model. Hence, for every extension of the *issues submodule*, there is a corresponding *actions submodule* extension. For example, when dealing with conformance issues (e.g., two classifiers with the same name), we will need to provide PARMOREL with a set of editing actions that could solve the issues (e.g., *delete* or *rename*).

5.1.2 *Preferences module*

Users can customize the results PARMOREL produces with their own preferences by implementing the *preferences module* (see Fig. 5.2). A preference indicates which kind of actions the user wants to apply in the models. When more than one action can be applied to solve an issue, the preferences are used to choose which one is best for the user. For example, if a user wants to produce models which are better with respect to a particular quality characteristic, PARMOREL would choose actions that have a positive impact on that characteristic. PARMOREL supports any type of preferences as long as they can be translated into numeric values (e.g., the value of a quality characteristic).

PARMOREL will take these values as rewards that will guide the repair process. These rewards will be used in the *learning module* to learn which action is the best to repair each issue. PARMOREL will use the rewards to estimate how good or bad each action is to satisfy the user's preferences.

The *preferences module* is independent of the *modeling module*, thus, a specific preference could be used regardless of the kind of issue being solved. For example, the same quality characteristics could be taken into account both for solving conformance issues and smells in the models. The only requirement is that they share the same supported models—Ecore class diagrams in the previous example.

5.1.3 *Learning module*

The *learning module* is responsible for learning which actions are the best to repair the issues in the models according to the preferences introduced by the users. It is also responsible for storing experience that can be used to streamline following repairs. The *learning module* is divided into two submodules, namely the *algorithm submodule* and the *experience submodule*, as showed in Fig. 5.2.

Algorithm submodule The *algorithm submodule* is the core of the learning process in PARMOREL and it contains the RL algorithm chosen for performing the repair. Within this submodule the process described in Section 4.2 takes place.

Experience submodule This submodule is in charge of storing and sharing experience in consecutive repairs by using TL, as explained in Section 4.3.

5.2 Extensions

In this section, we present the extensions we have created during the development of this thesis for each of the PARMOREL modules. These extensions work as an example

to explore what can be done with each module and to show the extensible potential of PARMOREL. However, the framework is not limited to these extensions, as PARMOREL is designed so that users can extend it to adapt it to their own repair requisites.

5.2.1 *Modeling module - Issues submodule*

We have developed the following extensions of the *issues submodule*: conformance issues, smells, and inter-model inconsistencies (see Fig. 5.3). Next, we present each of these extensions.

Conformance issues We have used the EMF diagnostician [88] to implement the identification of conformance issues that violate certain constraints of the Ecore metamodeling language [88] in Ecore class diagrams (e.g., the opposite of the opposite of a reference must be the reference itself, classifiers must have different names, etc). More details can be found in papers A, B and D.

Smells By using EMF [88] together with Edelta [23] we have implemented the *issues submodule* to be able to identify user-defined smells in Ecore class diagrams. Edelta is a model refactoring tool, based on a domain specific language (DSL) for defining Ecore model evolutions and refactorings. The core features of Edelta and its DSL have been detailed in [23]. By using the Edelta DSL, users can specify smells to be found in the models. More details can be found in Paper E.

Inter-model inconsistencies In this extension, we have used SDMetrics [94] to interact with UML models and implement a series of rules so that the *issues submodule* can identify inter-model inconsistencies. SDMetrics is an object-oriented design quality measurement tool for UML models. We use SDMetrics to analyze the models' structure and detect inconsistencies. This extension as well as *Repairs for inter-model inconsistencies* in Section 5.2.2 and *Coupling* in Section 5.2.3 are part of paper [12], which is in the second round of reviewing process in the SoSyM journal at the moment of writing this thesis (see Section 1.3). More details can be found in the Appendix.

5.2.2 *Modeling module - Actions submodule*

We have developed the following extensions of this submodule: repairs for conformance issues, refactorings for smells, and repairs for inter-model inconsistencies (see Fig. 5.3). Next, we present each of these extensions.

Repairs for conformance issues In order to repair conformance issues, we make use of the actions available within EMF to modify Ecore class diagrams. These actions implement operations of addition, removal, and updating of classes, references, attributes, and operations in the models. Some examples of the actions which are available in this extension are: *setName()*, *setOppositeReference()*, *removeSuperType()*, etc. More details can be found in papers A, B, and D.

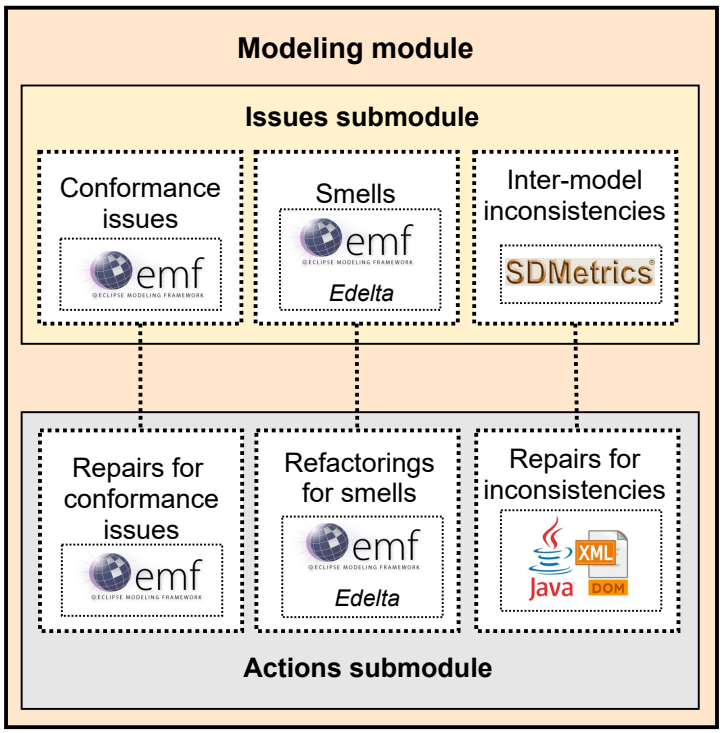


Fig. 5.3: Modeling module with sample action and issues submodules

Refactorings for smells In this extension, we use the Edelta DSL [23] together with EMF [88] to specify model refactorings. We define a smells resolver to resolve the smells found in the models. For example, the resolution for a smell with duplicated features can be managed by introducing a hierarchy (i.e., adding a super-class) and moving the shared features up to the newly created super-class. Unlike conformance issues, smells might be ignored and not removed since, sometimes, their removal might worsen the model's overall quality. Therefore, we also include an "ignore" action. More details can be found in Paper E.

Repairs for inter-model inconsistencies We use the Java DOM libraries to manipulate the UML models' structure as XML files. Unlike the previous issues, most inconsistencies have a single action to restore them, but the actions might be applicable in many different parts of the models. For example, if there is an operation in a class diagram without a corresponding message in a sequence diagram, there might be multiple potential senders and receivers, depending on the references the class containing the operation has to other classes. More details of this extension can be found in the Appendix.

5.2.3 *Preferences module*

We have implemented this module to support the following preferences: hard-coded preferences, quality characteristics, model distance, and coupling. Next, we present each extension.

Hard-coded preferences In initial versions of PARMOREL (see Paper A), we worked with hard-coded preferences. These preferences simulated different requirements a user might have about how to perform the repair, such as: avoid deletion of elements from the model, reward the repair of issues individually or in batch, and punish or reward modification of the original model structure.

For these preferences, the reward values were hard-coded by us. For example, some preferences provided a reward of 10 or a punishment of -10, depending on whether the applied action followed the chosen preference, multiplied by the number of times the action provided a result aligned or unaligned with the preference (e.g., if a user wanted to preserve the model original structure but an action created 5 new elements in the model, that action would receive a -50 as punishment). For other preferences, such as if the user preferred to repair issues individually, the actions would obtain a positive reward with the percentage of remaining non-repaired issues after repairing a single issue or a negative reward with the percentage of repaired issues.

These reward and punishment values were chosen according to our experimentation. In later papers, we decided to work with preferences of higher-level of abstraction and with a quantitative value associated, such as the rest of the preferences we present in this section. By using quantitative preferences, we did not need punishments anymore, as we got more diverse and accurate rewards, being higher the more a preference was satisfied.

Quality characteristics In this extension, we use quality characteristics extracted from the literature [25, 39, 76] as user preferences. PARMOREL integrates a quality evaluation tool which is inspired by [19].

This quality evaluation tool not only provides an evaluation mechanism but also supports the specification of custom quality characteristics. Calculation functions for each characteristic are implemented with the Epsilon Object Language (EOL) [54], an imperative programming language for creating, querying, and modifying EMF models. So far, we have specified the following quality characteristics to be used as user preferences: maintainability, understandability, complexity, and reusability. By using this extension, PARMOREL can learn how to repair Ecore class diagrams, regardless of the issues contained, in a way that the selected quality characteristics are improved. For more details about these quality characteristics, we refer to Paper B.

Model distance In Paper D, we exemplify an extension of the *preferences module* by using a model distance metric to guide the repair of Ecore class diagrams. PARMOREL obtains the distance metric from a model distance calculator. By using this metric we can reward the preservation of the original model structure when repairing, minimizing undesired side-effects in the repaired model. This model distance calculation is implemented as an Eclipse plugin, composed of a model matching algorithm specified with a customizable script which is written in the Epsilon Comparison Language (ECL) [53] script. More details can be found in Paper D.

Coupling By using the metrics offered in SDMetrics [94], we can define preferences to guide the repair of UML models. As an example, we use the metrics *MsgSent* (number of messages sent) and *MsgRecv* (number of messages received) [26] to calculate

the coupling in UML sequence diagrams. By taking into account the number of messages each lifeline in the sequence sends and receives, we can measure the degree of interdependence between the lifelines in the sequence, then we add these values to obtain the coupling of each lifeline. This way, users can decide to repair inconsistencies between class and sequence diagrams in a way that coupling in the sequence diagram remains as low as possible. More details about this extension can be found in the Appendix.

5.2.4 *Learning module - Algorithm submodule*

RL is a broad field with many algorithms. In Paper C, we have implemented this submodule to repair conformance issues in Ecore class diagrams while improving their maintainability with the following algorithms: Q(λ), Monte Carlo, SARSA and, SARSA(λ). As long as it implements the MDP concepts explained in Section 4.1, the algorithm will be supported by this submodule.

In Paper C, we take Q-Learning as the reference since it was the algorithm used in PARMOREL so far. In comparison, the algorithm that presented the best performance, both in time and number of episodes is Q(λ). The rest of the algorithms, Monte Carlo, SARSA(λ), and SARSA, perform worse than Q-Learning, being Monte Carlo the one with a better performance from this group, and SARSA the worse by a big difference. These results indicate the potential of Q(λ) for the model repair problem, while Monte Carlo, SARSA, and SARSA(λ) might not provide the best solutions.

5.2.5 *Learning module - Experience submodule*

This submodule has been implemented with our TL approach for model repair proposed in Section 4.3. More details about this submodule can be found in papers A and D.

This extension has been applied to repair conformance issues and smells in Ecore class diagrams, using as preferences quality characteristics and model distance metrics. We use the rewards stored from previous repairs to initialize the Q-table in the following repairs, reusing only those rewards relevant to the current user preferences. Previous repair processes influence the new repairs by jump-starting the repair process but do not interfere with learning new repair sequences. This way, the learning will converge faster and fewer episodes will be required.

CHAPTER 6

RESEARCH METHOD

In this chapter, we detail the research method we followed to develop this thesis. Additionally, we discuss the evaluation plan we followed to evaluate our work and how we selected the data for our experiments.

6.1 Constructive research

The research method followed during the development of this thesis has been constructive research [51, 60]. This method is a research procedure that aims at producing novel solutions to practically and theoretically relevant problems. It follows a pragmatic approach in which the ultimate goal is to achieve the solution of a problem by the construction of an artefact. This artefact, which is a combination of theory and practice, must be novel in its field. According to [48, 51] there are five main concepts within constructive research. Figure 6.1 details our results according to these concepts:

1. Theory connection: The theoretical knowledge, background, or state of the art of a field from which a problem or unexplored aspect in research can be identified. It is the foundation that motivates the problem to solve.

2. Practical relevance: The practical relevance that both the identified problem and the proposed solution will have to the field.

3. Construction: The novel artefact built during the research. This artefact provides a solution for the identified problem. The construction is constituted by theoretical and practical contributions.

   (a) Theoretical contribution: The novel theory created to support the construction. It provides a theoretical framework to solve the identified problem.

   (b) Practical contribution: The practical part of the construction. It focuses on the developed infrastructure, architecture, or software to handle the solution of the identified problem.

Following this research method, we started by studying the state of the art of the model repair field to identify relevant research problems and build our theory connection. The results of this research can be found in Chapter 2, where we detail model background, and in Chapter 8, where we analyze existing model repair approaches. After analyzing the current state and approaches within the field, we identified as

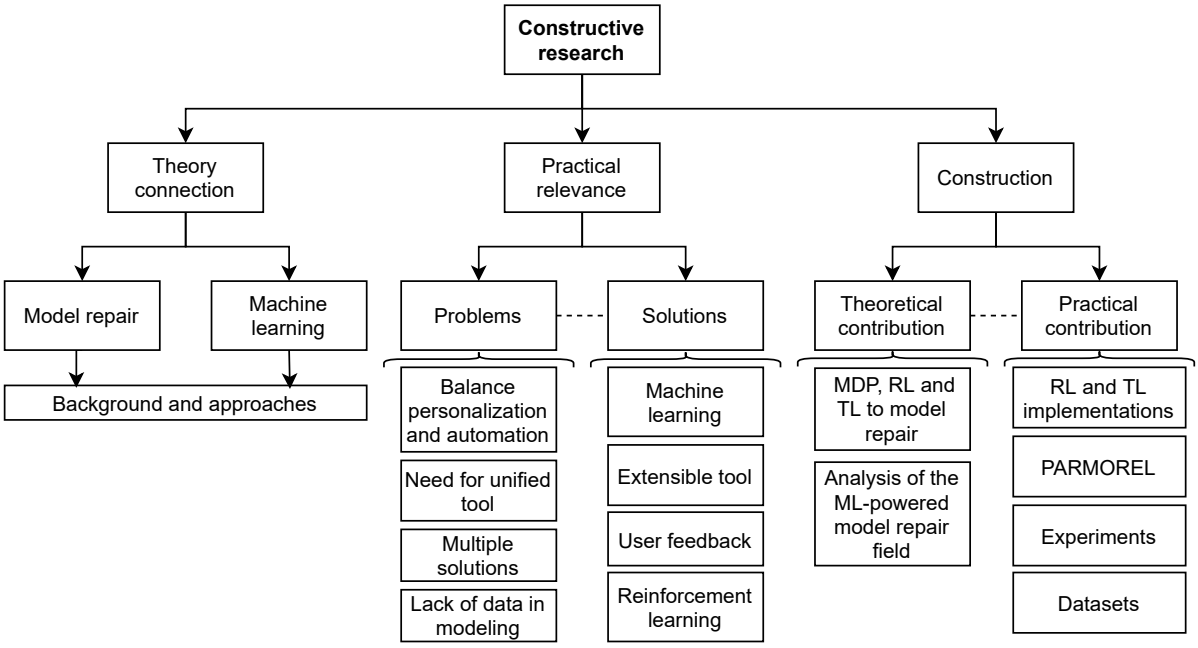


Fig. 6.1: Summary of our constructive research method process and its results

relevant problems: (i) the need for balance between automated and personalized approaches, (ii) the wide spectrum of issues, actions, and models existing within the field, and the lack of a unified mechanism to deal with them, and (iii) the multiple solutions that exist for repairing a single issue, needing some sort of user intervention to choose the most adequate solution.

Having identified these problems, and with the intention of building an artefact able to solve them, we started exploring the ML field, analyzing its existing approaches and thus, further developing our theory connection. The decision of researching ML methods was motivated due to our experience of working with ML before this thesis. We knew that ML had the potential to provide automation, personalization, and flexibility to address a wide variety of problems. From this research, we found a new relevant problem: the lack of data in the modeling field. Within the existing approaches in ML, we found we could solve this problem by using RL algorithms. The results of this research can be found in Chapter 3, where we detail ML background.

During our research, we could not find any work within the model repair field targeting these problems, nor applying tabular RL algorithms such as Q(λ). Hence, we could use these concepts to build a novel model repair artefact for solving the above-mentioned problems. First, we had to adapt the concepts of MDP, RL, and TL to the model repair problem, as we saw in Chapter 4, which led to our theoretical contribution.

On top of this theory, we built our practical contribution: implementations of RL and TL and the PARMOREL framework, as we described in chapters 4 and 5. For developing PARMOREL, we followed an iterative and incremental development, resembling the process of developing a minimum viable product [81]. We started by tackling the basics of our problems, such as achieving repair. This way, PARMOREL started as an approach to deal with conformance issues in Ecore class diagrams with a set of hard-coded user preferences. In further iterations, we evolved it by progressively

adding more features, such as adding more types of preferences at higher-level, issues, and types of models, and eventually providing the means so that users could add their own types themselves. Lastly, PARMOREL became an extensible framework that allowed personalized and automatic model repair. We could not find in the literature any research dealing with the model repair problem as an MDP nor providing our degree of customization. Some model repair approaches are independent of the domain of the models or support different model types, however, we could not find any approach presenting the degree of extensibility of our framework. Hence, PARMOREL is a novel artefact in the model repair field.

During this development process, we evaluated PARMOREL through a series of experiments in which we used different datasets. We consider these experiments and datasets also a part of our practical contribution.

6.2 Evaluation plan

In order to test our solution, our evaluation plan was oriented towards proving that our construction solved our identified relevant problems. This way, despite some other RQs raised during each experiment, our focus was always on evaluating whether: (i) the approach could provide automatic model repair, (ii) user preferences guided the repair, leading to personalized results, and (iii) our approach could handle different model repair elements, such as different types of models, issues, preferences, etc.

As mentioned in the previous section, our development process was iterative and incremental, thus new theory and implementations were built on top of the previous ones. This way, our evaluations followed a double-folded strategy, assessing whether our constructed theory was correct by developing a new version of PARMOREL which implemented new theory through new functionalities, and testing the correct functioning of the new version of PARMOREL. These evaluations were internal, carried in a controlled environment.

We followed a quantitative criteria for measuring whether our experiments achieved the objectives of our research. We analyzed if the repaired models were aligned with user preferences by analyzing if the numerical values of these preferences improved. Likewise, we addressed whether PARMOREL could support different types of models, issues, preferences, etc, by measuring if the issues in the models could be solved, counting how many of them were left in the models. Also, we performed scalability and performance experiments in which we measured the impact of the size of the models and the number of issues in the repair time.

6.3 Data selection

Since data is a crucial aspect of our experiments, we consider it relevant to state the criteria we followed to select our data. Our main option was to use data depicting real-world models, either coming from other existing works in the literature such as [10, 73] or from open repositories like GitHub or the ATL Zoo [5]. Then, we analyzed if these models contained the issues we were studying and if their formats were supported by the tools we integrated into PARMOREL (e.g., for calculating quality

characteristics).

In some cases, we needed to synthesize our own models, for which we used a random mechanism such as a mutation tool like AMOR [7], which introduced issues in Ecore models. For those situations where the format of the available models was unsupported by our tools or there were no available mutation mechanisms for creating the required issues, we opted to manually create the models, introducing issues inspired by those available in repositories or the literature.

CHAPTER 7

CONTRIBUTIONS

This chapter presents the contributions of our work. We start with a summary of the papers included as a collection in the Part II of this thesis. Then, we continue with an overview and discussion of the results obtained in the papers and conclude the chapter with the overall limitations and threats to the validity of our work.

7.1 Summary of papers

This section presents a summary of the papers produced during the development of this thesis. We present the papers in chronological order, as the research of each paper builds on the results of the previous ones.

7.1.1 *Paper A: Improving model repair through experience sharing*

This paper [18] was accepted at the Journal of Object Technology published in July 2020. In the paper, we emphasize that, although there are already approaches that provide an automatic repair of models, the same issues might not have the same solutions in all models due to different user preferences and business policies. Personalization would enhance the usability of automatic repairs in different contexts, and by reusing the experience from previous repairs we would avoid duplicated calculations when facing similar issues.

Here, we focus on repairing conformance issues in Ecore class diagrams. The main contribution of this paper is our TL proposal and an implementation to reuse the experience learned from each model repair. We explore TL's theory and adapt it to apply it to the model repair problem. We validate our approach by repairing models mutated with the AMOR tool [7] to include conformance issues in them. In the evaluation, we use different sets of personalization preferences and study how the repair time improved when reusing the experience from each repair.

We test our TL approach through two examples: first, we repair a broken model with different sets of preferences, and then, we repair 30 randomly mutated models obtained from 3 originals from GitHub. The objectives of this evaluation are to show that PARMOREL can (i) store and reuse experience learned from different preferences and (ii) improve the repair time when working with different models.

First, we simulate a set of seven users with different preferences to repair a single model, combining preferences such as rewarding/punishing modification of the

original model's structure and preferring the repair of issues individually or in batches. Some users have overlapping preferences (totally or partially) and opposite ones. This diverse set of preferences allows us to evaluate whether PARMOREL is able to: (i) share experience between users with unrelated preferences, (ii) successfully reuse experience when preferences coincide completely or partially with the stored experience, and (iii) achieve better performance when more parts of the experience are reused. The results of this repair are an indicator that PARMOREL allows to automatically store and share experience in different executions. Sharing is adapted depending on whether users introduce preferences already stored in the experience or not. By using TL, the repair time becomes faster when reusing more experience.

Then, to evaluate and test the generality and scalability of our approach, we repair 30 mutant models simulating 3 users with different preferences. The results of this evaluation indicate that our TL approach accomplishes sharing the experience learnt by repairing different models and streamlines the repair regardless of the chosen preferences.

7.1.2 *Paper B: Model repair with quality-based reinforcement learning*

Paper B was accepted at the Journal of Object Technology published in July 2020. This paper [49], is the first step in conceiving PARMOREL as an extensible framework. Here, we propose the integration of a tool to measure the quality of Ecore class diagrams based on different quality characteristics explored in the literature: maintainability, reusability, complexity, reusability, and relaxation. These quality characteristics are offered to users as preferences to guide the repair of the models, thus, a user may, for example, prefer a repair that prioritizes the maintainability in a model over other quality characteristics.

Unlike the previous paper (Paper A), here we evaluate the approach by repairing a dataset of 107 real-world Ecore class diagrams extracted from the literature, without using mutant models. In this paper, we study how the size of the models and the number of issues affect the repair time with and without TL, testing the scalability of PARMOREL. Also, we study if PARMOREL can select repair actions that boost the selected quality characteristics in the models. As user preferences, we choose to boost the maintainability of the models.

First, we configure PARMOREL to repair each model in 80% of the dataset. Our results indicate that both the size of the models and the number of issues affect the repair time logarithmically, although the influence of the latter is stronger. Then, we proceed to repair the remaining 20% of the dataset directly after repairing the previous 80%. Finally, we repair again the 20% after resetting the Q-table, that is, deleting the learning obtained from the 80% repair. By comparing the results from these two rounds, we can conclude that PARMOREL streamlines the repair time of the new models when it has learned from repairing other models.

The results of this evaluation indicate that PARMOREL is scalable and that it can handle real-world broken models. Furthermore, it is able to produce models aligned with the maintainability quality characteristic.

7.1.3 Paper C: A comparative study of reinforcement learning techniques to repair models

This paper [13] was accepted at the 2nd Workshop on Artificial Intelligence and Model-driven Engineering (MDEIntelligence), co-located at the ACM / IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS) in October 2020. In this paper, we formalize our RL approach for model repair as an MDP. Then, we revisit and improve our previous theoretical definitions of RL applied to model repair. We compare our old definition and our new one, namely, MDP-A and MDP-B by repairing a sample model. Our findings indicate that the MDP-B definition provides better results, hence, it is the one we integrate into PARMOREL for future repairs.

In previous papers (papers A and B), PARMOREL was powered by the Q-Learning algorithm. Here, we present as an alternative several other RL algorithms selected from [91]: Q(λ), Monte Carlo, SARSA, and SARSA(λ). We apply each algorithm to repair a set of models and compare the results of each algorithm in terms of repair time and episodes needed to complete the repair. Moreover, we measure if the repaired models present better maintainability. As a dataset, we use a representative sample of 12 models from the dataset used in [73], filtered in order to get only corrupted Ecore class diagrams.

Our results indicate the Q(λ) algorithm is the one that can repair with faster performance, while Monte Carlo, SARSA, and SARSA(λ) might not provide the best solutions. Moreover, with this experiment, we prove that PARMOREL can work with different RL algorithms.

7.1.4 Paper D: An extensible framework for customizable model repair

This paper [11] was accepted at the MODELS Conference in October 2020. In this paper, we present for the first time PARMOREL as an extensible model repair framework. Briefly, we present and explain PARMOREL modules, focusing on the preferences module.

Paper D presents an evaluation of PARMOREL focusing on the extensibility of user preferences. As an example, we extend the framework by including as a preference a model distance metric, which allows the user to choose a more or less conservative repair. We focus on evaluating if, by further extending the preferences, the framework is able to improve the precision in selecting better-repaired models. To achieve this, we implement as user preferences two different versions of the model distance metric: basic (coarse) and customized (fine-grained). We run PARMOREL first with the basic implementation of the distance calculation and then the customized one. To measure the impact that the two distance calculations have on the repair, we measure the following quality characteristics in the models: reusability, maintainability, complexity, and understandability. As dataset, we use the same models from Paper B.

With both distance calculations, the complexity improved or at least remained unchanged in all the models, but better results were obtained with the customized distance metric. Then, repairing the whole dataset, when using the customized metric, maintainability, reusability, complexity, and understandability improved in 80% to

100% of the models.

With the results of this evaluation, we can conclude that, by extending PARMOREL preferences, the precision of the framework improves and it is able to produce repaired models with higher quality characteristics.

7.1.5 *Paper E: Addressing the trade off between smells and quality when refactoring class diagrams*

This paper was accepted at the Journal of Object Technology in April 2021. In this paper [9], we demonstrate that PARMOREL can address different issues in models beyond conformance issues. We extend the issues and actions submodules to support a new type of issues and actions: smells and their corresponding refactorings. Removing smells sometimes might have a negative impact on some models, since by refactoring them the overall quality of the model could be worsened.

We implement the detection and selective refactoring of smells based on quality characteristics to assure that only the refactorings with a positive impact on the models' quality are applied. For the implementation, we select a representative sample of smells from the literature with their corresponding refactorings [24]. Furthermore, we present the changes we had to make to our RL approach to support smells detection and removal. Then, we analyze a dataset obtained from the literature [96] containing Ecore class diagrams, asserting that almost 90% of the models contained smells.

In this paper's evaluation, we check whether PARMOREL can decide which smells should be refactored in a model to maintain and, even improve, the quality characteristics selected by the user. For this evaluation, we use as dataset [96] which contains 555 Ecore class diagrams extracted from GitHub.

This evaluation indicates that, when taking into account the quality of the models, the best solution is usually not to remove all the smells. PARMOREL is able to refactor the models with a balance between which smells should be addressed without degrading the quality of the models and even improving it. In most cases, the refactored model presents even higher quality in the characteristics selected by the user than the original one. Furthermore, with this evaluation, we prove that PARMOREL is able to support different types of issues.

7.2 Overview and discussion of results

In this section, we present an overview of and discuss the results of our work. In this thesis, we have contributed to expand the modeling field by researching and applying, as a novel approach, RL algorithms to solve the model repair problem. We could not find any approach in the literature using the same algorithms and techniques as our research, hence, we have explored and opened a new research path in the intersection of ML and model repair.

PARMOREL presents a series of features we could not find in any other model repair approach. By creating a modular framework instead of a tool designed to tackle a specific problem, we have detached our approach from the problem to solve and overcome the challenge of lack of generality that most model repair tools present [71]. Our approach can be used to address the heterogeneity of stakeholders and

requirements existing in the model repair field by further extending its modules. Hence, PARMOREL could serve as a starting point for other researchers in the field to study different model repair scenarios with different learning algorithms.

Through the experiments in the above-mentioned papers, we have demonstrated that:

- Our approach provides automatic model repair applying concrete repair actions, such as renaming or deleting model elements. When the repair finishes, a repaired version of the original model is provided to the user as output.
- PARMOREL provides faster repair when facing issues already faced in other models due to the learning capability of RL. For example, in paper [15], we obtained an improvement of the repair time of $\approx 30\%$ when repairing models previously repaired.
- By using TL, we can streamline the repair regardless of the preferences chosen by different users. For example, in Paper A, we improved the repair time when transferring learning between users with different preferences between $\approx 10\%$ and $\approx 60\%$.
- By coordinating user preferences with RL rewards, PARMOREL can select the sequence of repair actions most aligned with the chosen preferences. As an example, we have used hard-coded preferences (see Paper A), quality characteristics (maintainability, reusability, understandability, complexity, and coupling) (see Paper B), and two different versions of a model distance metric (see Paper D).
- As long as it implements our MDP concepts for model repair, PARMOREL can work with different RL algorithms. As an example, we have tested the performance of Q-Learning, Q(λ), Monte Carlo, SARSA, and SARSA(λ). Q(λ) is the one that has provided better results when repairing models (see Paper C).
- By further refining user preferences, PARMOREL can produce higher-quality results. We demonstrated this by working with two different model distance metrics in Paper D.
- PARMOREL is able to repair different kinds of issues in the models. As an example, we have repaired conformance issues (see papers A, B, C, D), a set of smells extracted from the literature (see Paper E), and some inter-model inconsistencies between class and sequence diagrams (see Appendix).
- Likewise, different types of models are supported. As an example, we have worked with Ecore (see papers in Part II), UML class diagrams, and UML sequence diagrams (see Appendix).

With these results, we can conclude that PARMOREL is an automatic model repair approach that works with a balance between automation and personalization. Users not only can personalize the results of the repair but, since PARMOREL is designed as a modular framework, users can also extend and implement different modules to adapt the repair process to their requirements. PARMOREL allows users to (i) choose,

add and modify repair preferences, (ii) work with different types of models, (iii) edit which issues are detected, and (iv) with which actions they are addressed, as well as (v) customize the learning algorithm of the framework.

By using RL, PARMOREL is not limited to providing repair, but learns from it, which allows to improve the performance time the more models are repaired. And, by using TL, performance can improve despite the repair preferences chosen by different users.

PARMOREL is conceived as a framework that can be used by modelers to repair models in everyday modeling scenarios. The repair can be provided for a single or multiple models, without requiring any additional data beyond the models. Modelers can adapt PARMOREL to their specific repair needs and, if these needs change over time, modelers will not need to change to another tool, as they can extend PARMOREL modules to match their needs. PARMOREL repair is automatic, and with user intervention being optional beyond providing preferences before the repair, the framework is suitable for modelers with different levels of expertise.

7.3 Threats to validity and limitations

In this section, we discuss potential threats that are associated with the validity of our experiments. We distinguish between internal and external threats to validity. We present general threats, as each paper in Part II includes detailed threats for its experiments. Also, we discuss the limitations of our work.

7.3.1 Internal threats

Internal threats are factors influencing the outcomes of the performed experiments. Most of our experiments rely on external tools to provide user preferences [12]. Our smell detection and refactoring are also supported by an external tool [9]. In all these tools, preferences (quality characteristics and model distance) and smells together with their refactorings, are user-defined. To mitigate this threat, preferences, smells, and refactorings were reused, when possible, from existing definitions in the literature. Also, we represented the preferences and smells faithfully with the corresponding models, DSL syntax, or formulas.

Furthermore, our implementation of the learning module relies on a series of RL parameters (such as ϵ, α, λ, the maximum number of episodes, number of episodes for the early-stopping criteria, etc) [13]. Likewise, our distance metric calculation is parametric with respect to a match threshold [11]. Varying these parameters values may return different results. We tuned these parameters with different values until our experiments returned satisfactory results.

7.3.2 External threats

In this context, we discuss how the conducted experiments would still be valid outside the used settings. Most external threats in our research are related to the datasets used. For those experiments using mutant models, we rely on AMOR [15, 18] as an external tool to introduce random mutations in the models. Despite this randomness, it has

a predefined set of mutations, and the issues it produces might not be as complex as issues introduced by a human. Still, it is realistic to think these issues could appear in real modeling environments.

In some experiments, the amount of data used might not be considered large, but we mitigate this aspect by the heterogeneity of their sources; these models have been retrieved from different GitHub repositories and hence from different modelers or inspired by real-world models. Additionally, in most experiments, we rely on datasets used in other experiments in the literature, such as [73].

The models in the papers are based on Ecore and UML models (class and sequence diagrams), but we firmly believe it should be possible to switch to other model types by extending PARMOREL.

The set of preferences, types of issues, repair actions, and algorithms used in the papers are a representative sample of what can be found in the model repair and ML fields. Many other preferences could be measured in the models and, other issues could be identified together with different repair actions. We consider the set of issues, actions, and preferences representative enough since they are related to different elements in the models, covering a wide range of structural changes in them. Likewise, RL is a wide field with multiple algorithms. We have selected a set of five of them based on their suitability to solve the model repair problem.

7.3.3 Limitations

In the following, we present the limitations of PARMOREL. We plan to address these limitations as part of our future work.

Regarding personalization, no external users participated in the experiments. Instead, we simulated some user preferences and implemented them into the reward system.

At the moment, the experience submodule only supports dealing with Ecore class diagrams. This is because the issues we have tackled so far in UML models (inconsistencies between class and sequence diagrams) are more dependant on each model structure (creating or moving messages in different parts of the sequence) and their solutions are harder to generalize.

Currently, PARMOREL is limited to quantitative user preferences, as a numerical value needs to be provided as a reward. Also, PARMOREL needs to detect issues in a model in order to manipulate it, the approach cannot deal with models without issues yet.

Lastly, PARMOREL needs to get a set of actions to modify the model. Our approach works with the assumption that, for each issue in the model, between the actions available, there is at least one able to repair the issue. If this is not the case, PARMOREL will not be able to repair it, since, unlike other approaches, these actions cannot yet be inferred from the issues in the models. In this situation, PARMOREL will ignore the issue and continue with the repair process.

CHAPTER 8

RELATED WORK

In this chapter, we compare existing approaches for model repair with PARMOREL, hence putting our work into context. Model repair is a field with many different approaches and, since this thesis is about the intersection of ML and model repair, we will first compare PARMOREL with the most common trends among non-ML approaches, and then we will focus on those using ML.

We follow the same structure in all the subsections: first, we briefly mention the technique used to repair the models, followed by some approaches using the technique; then, we compare PARMOREL with the technique, and finally, we compare PARMOREL with the approaches.

8.1 Non-ML approaches

In this section, we follow the feature-based classification introduced by Macedo et al. [62] to present relevant model repair approaches. We classify approaches following the *Core* feature from [62], in which approaches are classified by the underlying technique they use for repairing models. According to Macedo et al., some approaches are hybrid, and hence built over several of these features (e.g., rule-based approaches that rely on search-based techniques to calculate repair plans from rules). Thus, the selection of features from this group is not exclusive.

Next, we classify PARMOREL according to the *Repair* features from Macedo et al. This classification and the claims we make throughout this section are true for the experiments we have performed on our datasets. Modifying the setting of our experiments might lead to different results. Following Macedo's classification, PARMOREL would be a *search-based* approach with *domain-specific procedures*. These procedures allow for finer control on the generation of repairs. PARMOREL achieves this by offering user preferences and allowing users to define their own actions. The repair representation is *operation-based* with *repair actions*, using *repair plans*, and applying *concrete operations*. In PARMOREL, repairs proposed to the user consist of a sequential composition of concrete edit actions, which can be directly applied to the model. These actions are *customizable* by users with the actions submodule. Regarding *enumeration* (the mechanism through which repairs are selected and presented to the user), the *output* can be *multiple*, presenting different repair options to the user when feedback is provided. The *order* (the set of the returned repairs and the order in which they are enumerated), is *parametrizable* and *interactive*. In PARMOREL, users can

control the behavior of the repair through preferences or by interacting with the tool at the end of the execution. Regarding *semantics*, as long as there exists actions able to repair the issues present in a model, PARMOREL is *total* (for every user update that results in an issue, PARMOREL is able to repair it), *correct* (when the repair is done, the model is free of issues), *well-behaved* (it assures to not create any new issues while repairing), *improving* (it reduces the number of issues in the model), and *fully consistent* (the number of issues present in the model is reduced to the minimum possible). Also, PARMOREL presents *stability* (it provides a null repair when there are no issues in the model) and *least change* repairs (when users select as preference a model distance metric, see Section 5.2.3).

In the following, we present rule-based, derivative, and search-based approaches, together with some representative examples found in the literature for each group. Then, we compare PARMOREL with these approaches.

8.1.1 Rule-based

Rule-based approaches rely on a set of pre-defined or extracted rules that are applied whenever an issue is detected in a model. These rules specify how issues must be solved.

In [72], Nassar et al. propose a rule-based approach that deals with violations of model instances with respect to Ecore models, from which the repair rules are automatically extracted. Using these rules, first, the approach checks if any parts of the model instance need to be trimmed (due to duplication or unnecessary information), and then, if there is any mandatory information missing (such as classes or attributes), it completes the instance. Once this process is completed, a valid model instance is generated. The authors have developed two Eclipse plugins, one for automatically deriving the repair rules from a given model, and another one with the repair algorithm taking as input these rules. Both the rules extraction and the repair algorithm can be executed automatically or interactively, with users being able to interrupt and guide the process.

Another rule-based approach is presented in [41], where the authors provide automated support for assisting designers in fixing issues in UML models. The consistency rules considered focus on keeping conformance in inter-related UML models. Here, users are not required to define repair rules, instead, the approach follows a trial-and-error process to generate values the model elements should take to solve the inter-model issues and where the issues should be solved. Both valid and non-valid options are generated and then pruned by the system. To deal with the scalability issues trial-and-error approaches usually present, the authors propose a custom-tailor generation of repairs. The final result is a set of choices on how to repair an issue that guarantees not to create any new issues in the model. The final repair is not automatically performed by the approach, the set of repair choices is offered to the user and it is up to her to choose which repair to apply. The approach is limited to repairs that involve a change in a single location at a time and to issues whose solution does not require introducing new model elements or new names. This approach is integrated within the design tool IBM Rational Rose.

The authors in [83] present CARE, standing for Constraint-bAsed REpairing of

ontological conformance relationships. CARE is based on the core concepts of Ecore (classes, attributes, references, etc) but it can deal with other modeling languages such as UML. The repair process is performed in three phases. First, CARE detects issues based on the conformance between a model and its metamodel. Additionally, the Object Constraint Language (OCL) and other user-defined constraints can be taken into account. Then, CARE extracts rules for repairing the detected issues, by modifying or deleting the faulty model elements. Finally, the extracted rules are ranked following configurable rules, based on heuristics, in which the user can manually incorporate structural as well as semantic knowledge. This ranking allows users to select the best repair according to their constraints. CARE depends heavily on the structural diversity of the classes in the metamodel. Consequently, if classes in the metamodel are structurally similar, CARE suffers from having a large solution space which increments the computational complexity of the repair process. Likewise, in certain cases, the presence of OCL constraints may lead to an empty solution, if no repair actions for OCL violations are present. Those repair actions have to be defined manually by the user beforehand.

In rule-based approaches in general, using rules has the advantage of providing full control over the resolution of issues achieving a tailor-made repair adapted to the need of each user. However, rule-based approaches require users to specify rules and constraints or select the source from which rules are extracted, requiring higher expertise from the user. Furthermore, by having a fixed set of resolution rules the flexibility of the approach is greatly reduced. When the repair rules are automatically extracted from a default third source (e.g. the Ecore metamodel), the expertise required from the user before the repair is reduced.

While rule-based approaches rely on a set of pre-defined or extracted rules that are applied whenever an issue is detected in a model, in PARMOREL there might be multiple actions that can potentially repair each issue. Hence, PARMOREL presents more flexibility than rule-based approaches. Furthermore, PARMOREL is not limited to repairing issues, as rule-based approaches are, but it is also able to learn how to repair them and to reuse that learning when facing the same or similar issues in the future.

Unlike rule-based approaches, PARMOREL does not offer a tailor-made repair, but users can choose their preferences so that the learning algorithm picks the actions most aligned with them. Moreover, users can provide feedback once the repair process is finished. By providing interaction prior or posterior to the repair, less time is required from the user than when defining rules or constraints. In rule-based approaches, the repair is performed directly, while in PARMOREL a solution must be discovered, and hence repair time will be longer initially. Using RL algorithms and TL compensates for this by learning from each repair and streamlining the repair process in consecutive repairs.

As in [72], users can interact with PARMOREL, but at the end of the repair process. We provide the interaction at the end of the execution so that users can take into account the consequences of each repair sequence in the model before choosing one of them. Choices are ranked based on the rewards obtained from the user preferences, similarly as ranking works in [83]. Approaches presented in [41] and [83] work as support systems where the repair choice is left to the users. In PARMOREL, users can

decide if they want an automatic or an interactive repair. PARMOREL is not tightened to a single modeling language like the approach in [83].

8.1.2 Derivative

Derivative approaches are those that automatically derive repair plans by analysis of the constraints imposed over a model (see *Syntactic* in [62]). Usually, repair plans in these approaches are calculated at static-time and then instantiated to concrete model instances at run-time when an issue is found.

In the literature, we can find derivative approaches like the one proposed by Xiong et al. in [95], where the authors propose Beanbag, a language to support the development of repair procedures. A Beanbag program defines and checks a consistency relation similarly to OCL, but it offers also the possibility to be executed in a fixing mode, taking user updates on the model and making the model satisfy the consistency relation. Instead of deriving fixing procedures purely from consistency relations or constraints, Beanbag allows users to define constraints and a fixing behavior at the same time, achieving a more tailor-made result in exchange for greater effort from the user. Beanbag is independent of the modeling language, as the authors test it with Ecore and UML models. As a limitation, Beanbag assures correctness of the results, but whether it produces or not an output depends on the procedures written by users. Also, when dealing with inconsistent data, the fixing functions might become too computationally expensive.

Another derivative approach can be found in [36], where Dam et al. present an approach to support change propagation within inter-related UML models without using hard-coded rules. Repair plans are generated when OCL constraints are specified by the user. Constraints are translated into a set of repair plans by following a systematic process defined by the authors. The repair is not performed automatically, as repair plans are offered to the users for them to choose from. Using this tool, users save time because they do not need to write repair rules and they only need to define OCL constraints.

Some derivative approaches present the advantage that they may be able to generate repair updates without user input, hence reducing the time required from the user prior to the repair but sacrificing personalization of repair. In this direction, PARMOREL requires few interaction from the user and, in exchange, it provides personalized results. As happened with rule-based approaches, PARMOREL goes beyond inferring and applying repair plans thanks to its learning potential.

In derivative approaches, the number of generated repair plans may become overwhelming for the user to choose from. In PARMOREL, unless the user wants to manually select a sequence of repair actions once the repair process finishes, the repair is guided by high-level preferences chosen prior to the repair. At the end of the repair process, PARMOREL will provide the user with the sequence of repair actions that satisfies the most her preferences.

Due to the number of repair plans generated, derivative approaches are not the best option when dealing with multiple issues or models with large portions corrupted. PARMOREL has no problem when dealing with multiple issues since although repair time might be slower the first time it faces such a model, the RL algorithm will be able

to learn a solution and provide faster repair when facing the same issues again. With respect to models with large portions corrupted, this is not a problem for PARMOREL since, by using our concept of issue, PARMOREL is only aware of what has to be repaired in the model and remains oblivious to the models' structure.

Regarding Beanbag [95], PARMOREL also assures correctness of the repaired models, but PARMOREL will always produce an output, independently of the extensions implemented by users. PARMOREL is also independent of the modeling language. Concerning [36], this approach works as a support system where users need to choose the repair to apply in the model, while in PARMOREL users can choose to do it this way or to apply the best repair according to their preferences automatically. Both [95] and [36] work with OCL constraints, hence repairing inconsistencies in model instances. In PARMOREL we work one level above, repairing issues originated from conformance issues between a model and its metamodel or from inter-model inconsistencies (see sections 2.2.1 and 2.2.3).

8.1.3 Search-based

Search-based approaches tackle model repair as a model search problem. This is, a problem defined by a set of states including a start and goal state, a boolean function that tells us whether a given state is a goal state, and a successor function able to create a mapping from a state to a set of new states. The start state in model repair will be the model with issues and the goal state the repaired model.

In [35] the authors propose an approach for generating repair plans for UML developers while they build their models. Plans are conformed by Praxis actions, a Prolog engine developed by the authors to detect violations of structural and methodological constraints specified on UML. In Praxis, it is assumed that model issues are introduced by user's actions that violate some of the model's constraints or by not executing actions required by these constraints. In order to generate a repair plan to fix these issues, Praxis first detects the actions that caused issues, then explores the possibilities for changing the inconsistent action, and finally generates a repair plan from the list of possible ways of changing the model. To detect the inconsistent actions, detection rules have to be manually defined in Praxis by the user. To deal with large search spaces, the approach includes a parameter with which users can select how much of the search space will be taken into account.

Another search-based approach is presented in [47], in which the authors focus on complementing Domain-specific modeling languages (DSMLs) with quick-fixes to maintain model consistency for complex language-specific constraints beyond the DSMLs scope. They propose a domain-independent framework, applicable to a wide range of DSMLs. The quick fixes are generated automatically, taking a set of domain-specific constraints and model manipulation policies as input. The approach uses graph transformation rules to specify the policies, which can be manually extended by users. The approach finds sequences of operations that decrease the number of issues. The authors test the approach in BPMN models.

In [61], Macedo et al. present Echo, a search-based approach to handle intra (such as conformance issues between a model and its metamodel or constraints related issues) and inter-model inconsistencies based on the relational model finder Alloy [50].

Echo works with Ecore as modeling language and with OCL for defining constraints. Inter-model consistency is specified by QVT Relations (QVT-R) or ATL transformation languages. Echo proposes repairs minimizing the model distance between the repaired model and the original one. The user is able to choose how to measure this distance: either through graph edit distance or through an operation-based distance. Echo also allows users to edit the repair actions. When the repair finishes, Echo presents all valid solutions, allowing the user to select the desired one.

Some search-based approaches rely on off-the-shelf solvers to search for the goal state [62]. These solvers are oblivious of the application domain and hence may produce unpredictable solutions. In contrast, other approaches rely on domain-specific search procedures [78], like heuristics and available edit actions, which allow a finer-grained control on the generation of repair updates. In general, search-based approaches are able to automatically fully repair models but have the disadvantage of suffering from scalability issues. Also, as an advantage, they are well-suited to fix issues that affect large parts of the model.

PARMOREL is powered by RL algorithms, which can be considered a search-based approach. The main difference between search-based and RL, apart from the learning potential offered by RL, is that search-based methods usually depend on a complete known problem. A complete known problem in terms of model repair means to know, for every issue and repair action available, what will be the outcome of applying the actions on each issue. This means, knowing beforehand which issues might be created while solving others and how the model structure will change during the repair process. When it comes to RL, usually the problem is partially unknown. There might be situations where it is impossible to predict what will be the effect of an action on an issue, as that may depend on other issues existing in the model and the model's elements. Likewise, in PARMOREL, the optimal actions chosen in each repair will vary depending on the preferences selected by the user.

RL usually performs faster than other search-based methods, as sometimes not the whole search-space is explored. The risk of not finding the optimal solution due to this situation is mitigated by adding a random factor such as ϵ (see Section 4.2). RL can be considered as an off-the-shelf solver, oblivious to the application domain and hence, might lead to unpredictable solutions. However, since in PARMOREL we take into account user preferences, the predictability of solutions increases, as they will be aligned with the preferences, hence allowing coarse-grained control from the user.

As search-based approaches, RL might suffer from scalability issues when dealing with very large models, however, although at first repair will be slower, in consecutive repairs it will be streamlined, hence mitigating this problem [49].

Regarding the Praxis approach [35], PARMOREL can deal with different types of models, while Praxis is limited to UML models. Praxis provides repairs by taking into account the actions applied in a model that led to creating issues, instead of targetting the issues directly as we do in PARMOREL. In Praxis, the detection rules are user-defined. This is similar to extending the issues submodule, however, in PARMOREL issues can be detected at a higher level by working, for example, with Ecore conformance issues by extending the issues submodule with the EMF diagnostician (see Section 5.2.1).

The work from [47] is different from PARMOREL, as this approach focuses on

providing quick-fixes in DSMLs. This approach guarantees to find a sequence of actions that decreases issues, but it does not deal with all issues present as PARMOREL does. In this approach, users can extend the transformation rules from which repairs are inferred, which is similar to extending the actions submodule in PARMOREL.

Echo is available for download at [46]. We have downloaded and tested Echo to compare it with PARMOREL. As in Echo [61], users can customize repair actions by extending the action submodule in PARMOREL. Likewise, both approaches provide concrete repairs, although in Echo it is mandatory that users manually select one of the repair options, and, in PARMOREL, this can be done without user intervention. PARMOREL provides the user with a list of the issues detected in the model and how each issue was addressed, while Echo does not provide any information to the user beyond the repair solution. Echo provides repair based on minimizing the model distance between the repaired and the original model. Although users can modify the model distance metric, Echo does not support other types of preferences (such as quality characteristics, coupling, or the definition of new preferences) as PARMOREL does. In Echo, the repair is performed step by step with user interaction (select repair preferences, decide where to apply the repair, and choose a repair solution), while in PARMOREL users only need to provide an input model and their repair preferences prior to the repair, with an optional interaction at the end of the repair process to choose a solution. Similar to PARMOREL, Echo supports the repair of different types of models as well as the restoration of intra and inter-model consistency. Finally, Echo has some functionalities that PARMOREL does not, like model generation, consistency checking, and model transformation, while Echo lacks the learning capability of PARMOREL.

8.2 ML approaches

Although ML can be considered an automated approach, not all automated approaches are ML, depending on whether they were tailor-made for solving a specific task or not. For example, rule-based [65, 72], graph-transformation [8, 64], and brute force [41] approaches cannot be considered ML due to their lack of generalization. Some rule-based approaches could be considered as rule-based ML when they are able to identify, generate, or modify rules on their own, however, approaches in the literature [55, 72] usually have as a pre-requisite some kind of definition or source of the rules.

In the following, we present an overview of the existing approaches of ML-powered model repair. We also include model refactoring since model repair can be seen as an activity that aims at resolving issues in models by refactoring them. The features presented by Macedo et al. [62] do not include learning aspects and ML is never mentioned in the classification. Hence, in this section, we follow our own classification of relevant ML model repair approaches, grouping them by the kind of learning algorithms they use.

Next, we compare PARMOREL with each of these approaches. Although some branches of RL have been applied to model repair (automated planning and genetic algorithms can be considered as RL), we could not find in the literature any approach tackling the model repair problem as an MDP nor using tabular algorithms such as Q-Learning or Q(λ).

8.2.1 Tree learning

The most extended approach is to use decision trees to support the repair. Decision trees are non-parametric supervised learning methods used for both classification and regression tasks. This approach goes from observations about an item to conclusions about the item's target value.

Kretschmer et al. introduce in [55] an approach for discovering and validating values for repairing issues automatically; they group alike these repair values if they have the same effect, which impacts positively the scalability of the approach. Prior to the validation, an input model with a set of consistency rules (such as OCL) is required. For each issue, a validation tree is constructed. The tree identifies all model elements involved (leaves) and shows how their values cause the issues. Then, using different techniques, the values on the leaves are changed, which may or may not solve the issues. Finally, by using boolean logic, the tree is analyzed to obtain which modifications solved the issues. The modifications found are concrete and can be executed automatically to repair the inconsistent model. Alike repairs are grouped and presented to the user as repair options.

Also tree-powered, Model/Analyzer [80] is a tool that, by using the syntactic structure of constraints, determines which specific parts of a model must be checked and repaired. A model with a series of constraints is required as input. The approach is independent of modeling and constraint languages, but the authors test Model/Analyzer on UML models with OCL constraints. This tool deals with a large number of repairs by focusing on what caused an issue and presenting repair actions as a linearly growing repair tree. The tree takes into account the side-effects of solving each issue, avoiding repairs that lead to new issues. At the end of the execution, issues are visualized, showing information about what parts of the model contributed to causing the issues and how to fix them. Knowing their origin may help users to solve the issues and prevent them in the future.

Lastly, Khelladi et al. [52] present a model repair approach that ranks repairs depending on the positive or negative side-effect they produce by using a validation tree. They also identify alternative repair paths and cycles of repairs. The approach works with UML models and OCL constraints. As output, a ranking of repairs is produced. Rather than an automatic approach, this approach can be considered as a guide and support system to assist modelers in solving model issues.

In tree approaches in general, as an advantage, the tree's nature allows reducing the state space size, hence optimizing the problem. Also, their structure makes them easy to understand and interpret, they require little or no data pre-processing, and unimportant features in the dataset will not influence the final result. However, trees tend to over-fit, producing results too closely related to a specific dataset. Their structure makes them unstable, as small changes in data can affect the structure of the tree and the final prediction, and inflexible, in the sense that to add any new data the tree should be retrained from scratch on the whole dataset.

RL and trees share the advantage of reducing the state space, making them suitable for optimization problems. PARMOREL, as happens in tree learning, does not require large amounts of data nor pre-processing in order to perform the repair, however, the results produced by trees are closely related to the dataset, and once the data change

they need to be retrained. By using RL, PARMOREL can work with different datasets learning on the run and reuse what was learnt in previous repairs. Likewise, being less dependent on the dataset, PARMOREL produces more stable results.

Due to their exploitation nature (probing a limited region of the search space), trees tend to lead to the same solutions once and again. Differently, the RL algorithms in PARMOREL include both exploitation and exploration, allowing to find new and, sometimes, better solutions for a given issue.

Approaches using tree learning such as [52, 55, 80], usually work as support systems, leaving the final repair decision to the user. While this provides a finer-grained control, it requires more time from the user. PARMOREL is not designed to be a support system, as we take advantage of the learning capabilities of RL to provide automatic repair.

8.2.2 Automated planning

Automated planning is an AI technique that focuses on the optimization of sequences of actions. Unlike classical classification and control problems, planning solutions are complex and have to be discovered and optimized in a multidimensional space. Planning can be classified as a branch of RL [56]. By using automated planning, it is possible to generate plans that lead from an initial state to a defined goal.

Puissant et al. present Badger in [78, 79], a tool based on automated planning. In Badger, the initial state is the state of the model prior to the repair process. Each input model is defined as a series of elementary operations which are needed to create the model (e.g., *create, addProperty, addReference*, etc.). The operations include as parameters the author of the operation and versioning information of the model. By using this format, the elementary operations become metamodel independent, and they can be used together with different kinds of structural metamodels. The defined goal is the set of consistency rules that need to be satisfied in the input model. By default, Badger chooses the minimum repair actions to reach the defined goal, however, users can modify this to prioritize specific types of actions or parts of the model to repair. Their approach is applied to different types of structural issues in UML models.

Badger is metamodel-independent and its planner algorithm can be adapted to work in different domains. To show this, the authors create a metamodel to represent and repair Java code smells. The planner does not require the user to specify resolution rules manually or to specify information about the causes of the issues, hence the tool is fully automated.

Automated planning presents advantages and disadvantages similar to those presented in search-based approaches. Automated planning can be considered as an off-the-shelf solver, independent of the application domain, and hence may produce unpredictable solutions. Likewise, due to the exploration of the state space that planning performs, it might suffer scalability problems when dealing with very large models.

The comparison between automated planning and RL is very similar to the one we described with respect to search-based approaches in the previous section. Automated planning usually deals with complete known problems. It is only possible to create plans when knowing the environment of a problem, whether it is the real one or an estimated one, like a simulation. However, in RL, sequences are learnt by trial and

error via interactions with the real environment, which is not completely known.

By using automated planning, Badger [78, 79] generates plans that lead from an initial state to a defined goal, each plan being a possible way to repair one issue. In PARMOREL, we prefer to repair the whole model since some actions might modify the model drastically, and we consider it counter-intuitive to decide which action to apply on an issue without knowing its overall consequences. This is the reason why we give rewards when all issues have been addressed at the end of an episode. Furthermore, PARMOREL is able to streamline the repairs and reuse what was learnt from previous repairs, while Badger's repair time remains static.

While Badger is metamodel-independent, PARMOREL goes one step beyond, as the type of models, issues, actions, and preferences can be modified. Regarding preferences, Badger allows to prioritize certain types of actions or to focus on repairing specific parts of the model. In the initial versions of PARMOREL, we included these types of preferences (see Paper A), but in later versions, we preferred to focus on higher-level preferences such as quality characteristics (see Paper B). Lower-level preferences might have the risk of providing a repair which consequences can not be measured beforehand by the user, for example, by rewarding certain types of actions without knowing how they will modify the model.

8.2.3 Neural networks

Some approaches make use of neural network (NN) architectures. NNs are an ML architecture, and there exist different types of NNs able to solve problems within supervised, unsupervised, and reinforcement learning. NNs are inspired by the brain's structure and simulate a net of neurons able to identify patterns and correlations in data. NNs contain multiple layers of a data structure called neurons, which are connected with each other. These connections have changing weights that simulate the connection of neurons in the human brain. Connections with stronger weights will be favoured and will lead to learning the solution for a given problem. These algorithms allow to perform multiple tasks given a dataset with enough examples from which the network can learn.

In [29] the authors present a NN for model transformation without specifying code for any specific transformations. Although not specifically model repair, we consider this approach close enough to be included in this section. They make use of a dataset of UML models generated by a Java program, but the approach is open for other modeling languages. Models are stored in a tree structure using JSON formatting, the root contains the keyword MODEL, and its children are the model elements. Then, the network transforms the input models into their corresponding output models extracting the model transformation needed.

Tackling model refactoring, in [87] the authors make use of a deep NN architecture to refactor UML diagrams with symptoms of design flaws. In this approach, the deep NN learns to recognize the presence of functional decomposition in UML models of object-oriented software, producing as output a refactored model without flaws. They use a dataset comprising feature vectors of distinct UML class diagrams. They obtained the dataset from the UML-Ninja repository [3] and extracted metrics from them by using SDMetrics [94]. In both approaches, the authors claim their results are promising

but there are still a series of open challenges needed to be addressed, such as the size of the training dataset, diversity of data, generality, etc.

The main advantage of NNs approaches is that, once trained, they produce results very fast and they can solve a wide range of problems. However, they need a great amount of data to be able to solve a problem. NNs depend a lot on training data which may lead to over-fitting and generalization problems. Lastly, NNs work as black boxes, meaning it is hard to know why a specific result was obtained and which variables influenced it.

NNs and RL are designed to deal with different problems. NNs usually deal with regression and classification problems by being fed with great amounts of pre-processed data, while RL algorithms learn how to solve a problem in an environment by interacting with it, without needing training data. Current modeling repositories are still limited in terms of size, labeling, and diversity of models. Hence, the lack of data is a challenge for ML adoption in modeling problems like model repair [27, 29, 30, 45, 85]. Due to this data challenge, RL algorithms present an advantage to NNs when solving the model repair problem.

The solutions provided by NNs are tightly related to the training dataset; if the requirements of the problem change, the data also needs to change (as happened in tree learning). By using RL in PARMOREL, the algorithm learns by directly interacting with the models and, by using the abstract concepts of issue, action, reward, etc, our approach can easily be adapted to solve different problems without the burden of designing new datasets.

Although RL algorithms can also be considered as black boxes, the origin of their results is easier to interpret than in NNs. In PARMOREL, the chosen actions will be those that maximize the rewards obtained from user preferences, thus, actions most aligned with preferences tend to be chosen. It is possible to backtrack how Q-values were obtained from rewards and knowing why one action was preferred over the rest.

Following the approaches in [29] and [87], PARMOREL modules could be extended with NNs, both to identify issues and repair actions from the models and to learn how to repair them according to preferences. As in [87], we also use SDMetrics [94] in PARMOREL, in our case to calculate coupling in UML sequence diagrams (see Appendix).

8.2.4 Genetic algorithms

Genetic algorithms are used to generate solutions to optimization and search problems by using biologically inspired operators such as mutation, crossover, and selection. These algorithms fall under an AI branch called evolutionary algorithms, however, since they aim to solve similar problems as RL algorithms (searching for solutions that maximize or minimize a reward or cost function) and behave in a similar way (finding a solution by interacting with an environment with no training data), genetic algorithms can be considered a branch of RL [67].

In [45], the authors present an approach based on an interactive genetic algorithm. In this paper, the authors make use of a fitness function that combines the similarity between the analyzed design model and models from a base of examples, and the modelers' feedback. Modelers introduce their feedback after some iterations of the

algorithm, indicating which solutions from the ones found they prefer. Users can specify different parameters, such as the percentage of solutions shown in each interaction. Hence, some expertise in these algorithms is required. The tool takes as input a base of examples of refactored models and an initial model to refactor, then, it generates as output a sequence of refactorings to be applied on the input model. This tool is developed as an Eclipse plugin. As a dataset, they use Ref-Finder [84] to extract refactorings performed in different Java projects, working with UML models. In this approach, the feedback from the user modifies how the genetic algorithms work and find repairs.

Genetic algorithms have the advantage that, if used properly, they may produce a solution faster than other algorithms. Likewise, their concepts are easier to understand than other algorithms, as they follow bio-inspired mechanisms and use vectors as data structures. These algorithms start with a population of randomly generated solutions and use the principle of natural selection to discover useful sets of solutions by combining and updating the population. The nature of genetic algorithms is based on heuristics and random updates of the population of solutions. This randomness might lead to situations in where the algorithm is not always able to find the optimum solution to a problem or gets stuck in local maximum and fails to find a suitable solution at all. This situation might be mitigated by adding more iterations to the algorithm, however, it will also increase the running time.

Genetic algorithms are used to generate solutions to optimization and search problems, looking to maximize a fitness function. They work in a similar way to RL, where the goal is to maximize a reward. The nature of genetic algorithms is based on heuristics and random updates of the population of solutions. RL algorithms in PARMOREL are ϵ-greedy and hence also rely on a random component, however, Q-values updates are not arbitrary, as they follow a gradient-based update using the Bellman's equation [91].

Although genetic algorithms tend to produce solutions faster, PARMOREL presents some mechanisms not supported by genetic algorithms to mitigate this situation, such as streamlining learning by reusing what was learnt in previous repairs. Likewise, the experience acquired in PARMOREL when repairing different models can be reused while in genetic algorithms, populations can hardly be reused.

Regarding [45], where user interaction is necessary for the tool to work, in PARMOREL, interaction is optional and happens at the end of the repair process, allowing users to choose from the found repairs. If users want to modify how PARMOREL learns, they can use preferences prior to the repair.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This chapter concludes the Part I of this thesis. We begin by re-visiting the RQs presented in Chapter 1 and summarizing the main contributions of the thesis. To conclude, we outline directions for future work.

9.1 Research questions revisited

The main research focus of this thesis was on applying RL to model repair and developing an approach to achieve personalized and extensible model repair. In this section, we re-visit our RQs and summarize our answers to these questions according to the results obtained in this thesis.

RQ1: How can RL algorithms improve model repair? How can we apply RL algorithms in model repair?

RQ1 focused on investigating RL algorithms: how they can be applied to model repair and how useful they can be in solving current model repair challenges. As stated in Section 6.1, we identified as relevant problems in the model repair field: (i) the need for balance between automated and personalized approaches, (ii) the wide spectrum of issues, actions, and models existing within the field, and the lack of a unified mechanism to deal with them, and (iii) the multiple solutions that exist for repairing a single issue, needing some sort of user intervention to choose the most adequate solution.

Due to its ability to adapt to data and different scenarios, RL can provide automated solutions while adapting to the needs of the users. Likewise, these algorithms present flexibility to solve different problems and to deal with the different types of models, issues, etc, that are part of the model repair problem. Also, several mechanisms are supported so that users can interact and provide their feedback to these algorithms.

We discarded most ML algorithms due to the lack of data available in the modeling field and the nature of the data existing on current modeling repositories (small variety, poor quality, unlabeled). Hence, due to this data issue, we chose to apply RL to solve the model repair problem. RL algorithms are a solution that allows personalization of results without needing large amounts or pre-labeled data. To apply RL to solve the model repair problem it was first necessary to characterize how this problem is defined within RL terms. As a result, we formalized the model repair problem as an MDP and developed our theory for applying the notions of RL and TL to the model repair problem.

RQ2: How can we keep a human in the loop when performing model repair? How well can human intervention improve results?

RQ2 focused on exploring and identifying which mechanisms are most beneficial to keep a human in the loop while repairing models and how this interaction affects the final results.

For answering this RQ, we focused on the concept of reward and how we could adapt it to align it to the user requirements. The answer was to develop the preferences module within the PARMOREL framework, where users could define their own repair preferences as long as these could be quantified and hence, used as rewards in the RL algorithm. This way, the algorithm would eventually choose repair actions aligned with the user preferences, leading to personalized results. According to our experiments, by further extending preferences, models with better quality characteristics can be obtained.

Moreover, for those situations where preferences were not enough, we developed a mechanism through which users could provide their feedback at the end of the repair process. This way, users could manually select the sequence of repair actions they preferred, and their feedback would override the rewards of the algorithm, hence being remembered in future repairs.

Also, we have defined our own TL approach for model repair, with which the preferences of different users and what the algorithm learns with their repairs can be forwarded to future repairs, hence taking advantage of what was already learnt from other users to streamline the repair process.

RQ3: How can a model repair framework be designed to tackle the variety of repair situations existing in the model repair field?

RQ3 focused on researching how the outcomes of **RQ1** and **RQ2** could be incorporated into a model repair framework that tackles the wide range of problems that can be solved in the model repair in a unified way.

Instead of having a specific tool for each kind of issue and each type of model, we propose a unified extensible framework: PARMOREL, which can be extended to support the new requisites that modelers need to address in their model repair processes. Powered by RL, PARMOREL focuses on supporting personalization of repair results and on extending its functionality to provide repair for different kinds of models, issues, and repair preferences. Likewise, it is possible to modify the editing actions and the RL algorithm used.

PARMOREL follows a modular structure, divided into three main modules: modeling, preferences, and learning module. PARMOREL is designed as an Eclipse plugin and users can implement its modules through a series of interfaces. We have demonstrated PARMOREL's extensibility by extending each of these modules with a series of extensions (see Section 5.2). The framework is not tied to these extensions and hence, could be further extended.

9.2 Summary of contributions

In this section, we summarize the contributions of this thesis by dividing them into theoretical and practical contributions, according to the constructive research method we detailed in Chapter 6.

9.2.1 *Theoretical contribution*

We consider as a theoretical contribution the theory built during the development of this thesis. Our theoretical contribution is related to adapting RL to solve the model repair problem. This theory is novel, and according to our research, no other work has built theory in this direction. We detail our contribution in Chapter 4.

First of all, we have formalized the model repair problem as an MDP, defining abstract concepts such as states, actions, and rewards that can be implemented using different modeling elements (e.g., a state can correspond with different types of issues). Then, we have defined how to use RL, specifically some tabular algorithms such as Q(λ), to solve the model repair problem according to the above-mentioned MDP definition. Lastly, we have defined our own TL approach, by combining existing methods (starting-point and imitation methods, see Section 3.5), that allows us to reuse the experience obtained from previous repairs to streamline the next repairs.

9.2.2 *Practical contributions*

The main practical contribution of the thesis is the PARMOREL framework. The framework solves the problems we identified in the literature (see Chapter 6) and gives answers to our RQs. The PARMOREL framework is detailed in Chapter 5.

The implementations for RL and TL used by PARMOREL are detailed in Chapter 4. We provide details about both implementations so they can be reproduced by other researchers. Furthermore, both implementations are based on abstract concepts that can be implemented in various ways, hence, our approach could be adapted to solve different problems.

The experiments performed in our evaluations, together with their respective outcomes were summarized in Chapter 7. These experiments are used as a proof of concept to evaluate our approach and it is still pending to evaluate it with modelers in real-world scenarios. We provided enough data for other researchers to replicate our experiments and compare their results to ours. Likewise, our datasets are publicly available in [2], so that they can be freely used for experimentation. By providing these datasets we contribute to improving the lack of data in the modeling field.

9.3 Future work

The work presented in this thesis provides several directions for future work given the extensible nature of PARMOREL, its current limitations, and the identified challenges in the ML-powered model repair field.

9.3.1 *Overcoming limitations*

First, PARMOREL could be extended to overcome its current limitations. Now, PARMOREL focuses on repairing or solving issues contained in models. In this direction, new mechanisms could be integrated so that the framework could learn how to improve models without issues, for example, by boosting their quality characteristics.

Also, PARMOREL could be provided with the means to, in a similar way to other approaches such as derivative (see Section 8.1), infer repair actions automatically from issues present in the model, removing the need for providing the framework with a set of repair actions. Likewise, the experience submodule could be extended so that TL can be applied in more complex scenarios beyond the repair of conformance issues.

At the moment, PARMOREL works sequentially and concurrent sharing of experience is not supported. That is, we store experience in an XML file that can be shared via a repository. This method works as a proof of concept, however, a collaborative environment could be created, where experience is gathered and shared at runtime. Furthermore, an option could be provided so that users can share their own preferences and explore other users' preferences. In this direction, PARMOREL could be extended to work with collaborative modeling environments, where PARMOREL would be able to detect issues as they happen, forwarding the repair to all the users involved.

9.3.2 *Further evaluation*

In this thesis, we have claimed that PARMOREL is able to support different types of learning algorithms, which we have evaluated by repairing models with a set of different RL algorithms (see Paper C). To further support this claim, the learning module could be extended with other RL and ML algorithms.

Likewise, it is pending an evaluation where real modelers of different levels of expertise use PARMOREL, both with our current modules extensions and by asking them to implement their own. This way, it would be possible to evaluate the extensibility potential of PARMOREL in a real setting.

In Chapter 8, we have compared PARMOREL with other model repair approaches. However, we have not applied PARMOREL to repair the same models as other approaches. Hence, there is room for a more exhaustive comparison, studying parameters such as repair time or the quality of the models produced. In this direction, a benchmark could be created by using different modeling datasets, including the ones used in the papers of the Part II of this thesis. By using this benchmark, PARMOREL's results and its performance could be compared to other existing model refactoring and repair approaches in the literature. Due to their tailor-made nature, some approaches are currently difficult to compare with others, hence, it will be necessary to find objective evaluation metrics to compare them.

Lastly, an interesting research line would be to explore the potential of PARMOREL to develop models from scratch in inter-related model scenarios, as explained in the Appendix. This could be evaluated in projects containing different UML models, providing PARMOREL with one of the models of the project and evaluating how well it can create the rest of the models.

BIBLIOGRAPHY

[1] OMG unified modeling language specification, version 2.5.1 (2017). Last accessed on 22/05/2021, https://www.omg.org/spec/UML/. 2.1.1, 2.1.2

[2] Project PARMOREL - HVL research in information and communication technology blog. Supported by Western Norway University of Applied Sciences, last accessed on 19/05/2021, https://ict.hvl.no/project-parmorel/. 1.3, 5.1, 9.2.2

[3] UML-Ninja. Last accessed on 24/03/2021, http://models-db.com/. 8.2.3

[4] L. Addazi, A. Cicchetti, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Semantic-based model matching with emfcompare. In *10th Workshop on Models and Evolution (ME 2016)*, pages 40–49, 2016. 1

[5] F. Allilaire. ATL Transformations. Last accessed on 24/03/2021, https://www.eclipse.org/atl/atlTransformations/. 4, 6.3

[6] E. Alpaydin. *Introduction to machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, third edition, 2014. 3.1, 3.2

[7] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR–towards adaptable model versioning. In *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*, volume 8, pages 4–50, 2008. 6.3, 7.1.1

[8] C. Amelunxen, E. Legros, A. Schürr, and I. Stürmer. Checking and enforcement of modeling guidelines with graph transformations. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, pages 313–328. Springer, 2007. 8.2

[9] A. Barriga, L. Bettini, L. Iovino, A. Rutle, and R. Heldal. Addressing the trade off between smells and quality when refactoring class diagrams. *Journal of Object Technology*, 20(3):1:1–15, June 2021. The 17th European Conference on Modelling Foundations and Applications (ECMFA 2021). 1.2, 7.1.5, 7.3.1

[10] A. Barriga, D. Di Ruscio, L. Iovino, P. T. Nguyen, and A. Pierantonio. An extensible tool-chain for analyzing datasets of metamodels. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–8, 2020. 4, 1.3, 6.3

[11] A. Barriga, R. Heldal, L. Iovino, M. Marthinsen, and A. Rutle. An extensible framework for customizable model repair. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 24–34, 2020. 1.2, 7.1.4, 7.3.1

[12] A. Barriga, R. Heldal, A. Rutle, and L. Iovino. PARMOREL: A framework for customizable model repair. Manuscript submitted for publication in the International Journal on Software and Systems Modeling (SoSyM). Submitted version available at: https://bit.ly/3xSCRVs, 2021. 5, 1.3, 5.2.1, 7.3.1, A

[13] A. Barriga, L. Mandow, J. L. P. de la Cruz, A. Rutle, R. Heldal, and L. Iovino. A comparative study of reinforcement learning techniques to repair models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–9, 2020. 1.2, 7.1.3, 7.3.1

[14] A. Barriga, A. Rutle, and R. Heldal. Automatic model repair using reinforcement learning. In *International Workshop on Analytics and Mining of Model Repositories (AMMoRe), co-located with MODELS*, pages 781–786, 2018. 1, 1.3

[15] A. Barriga, A. Rutle, and R. Heldal. Personalized and automatic model repairing using reinforcement learning. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 175–181, 2019. 2, 1.3, 7.2, 7.3.2

[16] A. Barriga, A. Rutle, and R. Heldal. Towards quality assurance in repaired models with parmorel. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD), JISBD 2019 (Cáceres), ISDM: Ingeniería del Software Dirigida por Modelos*, 2019. 3, 1.3

[17] A. Barriga, A. Rutle, and R. Heldal. AI-powered model repair: State of the art, challenges and opportunities. Manuscript submitted for publication in the International Journal on Software and Systems Modeling (SoSyM). Submitted version available at: https://cutt.ly/ObTf3cX, 2021. 6, 1.3

[18] A. Barriga, A. Rutle, and H. Rogardt. Improving model repair through experience sharing. *Journal of Object Technology*, 19(2):13:1–21, July 2020. 1.2, 7.1.1, 7.3.2

[19] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. A customizable approach for the automated quality assessment of modelling artifacts. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 88–93. IEEE, 2016. 2.3.1, 5.2.3

[20] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. A tool-supported approach for assessing the quality of modeling artifacts. *Journal of Computer Languages*, 51:173–192, 2019. 1, 2.3.1

[21] M. Bats. UML Designer. Last accessed: 2021-06-08, http://www.umldesigner.org/. A

[22] K. Beck, M. Fowler, and G. Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, 1:75–88, 1999. 2.2.2

[23] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio. Edelta: An approach for defining and applying reusable metamodel refactorings. In *MODELS (Satellite Events)*, pages 71–80, 2017. 5.2.1, 5.2.2

[24] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio. Quality-driven detection and resolution of metamodel smells. *IEEE Access*, 7:16364–16376, 2019. 2.2.2, 7.1.5

[25] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976. 5.2.3

[26] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In *Proceedings of the 19th international conference on Software engineering*, pages 412–421, 1997. 5.2.3

[27] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, 2020. 1.1, 8.2.3

[28] A. Bucchiarone, F. Ciccozzi, L. Lambers, A. Pierantonio, M. Tichy, M. Tisi, A. Wortmann, and V. Zaytsev. What is the future of modeling? *IEEE Software*, 38(2):119–127, 2021. 1

[29] L. Burgueño, J. Cabot, and S. Gérard. An LSTM-based neural network architecture for model transformations. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 294–299. IEEE, 2019. 1, 8.2.3

[30] J. Cabot, R. Clarisó, M. Brambilla, and S. Gérard. Cognifying model-driven software engineering. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 154–160. Springer, 2017. 1, 8.2.3

[31] F. Cady. Machine learning classification. In *The Data Science Handbook*, pages 97–120. John Wiley & Sons, Incorporated, New York, 2017. 3.1

[32] A. A. Cervantes, N. R. van Beest, M. La Rosa, M. Dumas, and L. García-Bañuelos. Interactive and incremental business process model repair. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 53–74. Springer, 2017. 1

[33] K. Chandra, G. Kapoor, R. Kohli, and A. Gupta. Improving software quality using machine learning. In *Innovation and Challenges in Cyber Security (ICICCS-INBUSH), 2016 International Conference on*, pages 115–118. IEEE, 2016. 1

[34] R. Chang, S. Sankaranarayanan, G. Jiang, and F. Ivancic. Software testing using machine learning, Dec. 30 2014. US Patent 8,924,938. 1

[35] M. A. A. Da Silva, A. Mougenot, X. Blanc, and R. Bendraou. Towards automated inconsistency handling in design models. In *International Conference on Advanced Information Systems Engineering*, pages 348–362. Springer, 2010. 8.1.3

[36] H. K. Dam and M. Winikoff. Supporting change propagation in UML models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010. 8.1.2

[37] R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software technology*, 50(12):1281–1294, 2008. 2.1

[38] M. Dirix, A. Muller, and V. Aranega. Genmymodel: an online UML case tool. 2013. A

[39] R. G. Dromey. A model for software product quality. *IEEE Transactions on software engineering*, 21(2):146–162, 1995. 5.2.3

[40] A. Egyed. Instant consistency checking for the UML. In *Proceedings of the 28th international conference on Software engineering*, pages 381–390, 2006. A

[41] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 99–108. IEEE, 2008. 8.1.1, 8.2

[42] S. Feldmann, K. Kernschmidt, M. Wimmer, and B. Vogel-Heuser. Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. *Journal of Systems and Software*, 153:105–134, 2019. 2.2.3

[43] A. Ferdjoukh, F. Galinier, E. Bourreau, A. Chateau, and C. Nebut. Measuring differences to compare sets of models and improve diversity in MDE. In *ICSEA: International Conference on Software Engineering Advances*, 2017. 2.3.2

[44] O. Friedrichs, A. Huger, and A. J. O'donnell. Method and apparatus for detecting malicious software through contextual convictions, generic signatures and machine learning techniques, July 21 2015. US Patent 9,088,601. 1

[45] A. Ghannem, G. El Boussaidi, and M. Kessentini. Model refactoring using interactive genetic algorithm. In *International Symposium on Search Based Software Engineering*, pages 96–110. Springer, 2013. 1, 8.2.3, 8.2.4

[46] Haslab. haslab/echo. Last accessed on 12/07/2021, `https://github.com/haslab/echo`. 8.1.3

[47] Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró. Quick fix generation for DSMLs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 17–24. IEEE, 2011. 8.1.3

[48] R. Hyötyläinen, K. Häkkinen, and K. Uusitalo. The constructive approach as a link between scientific research and the needs of industry. In *Proceedings of the Logistics Research Network Conference*, 2014. 6.1

[49] L. Iovino, A. Barriga, A. Rutle, and H. Rogardt. Model repair with quality-based reinforcement learning. *Journal of Object Technology*, 19(2):17, 2020. 1.2, 7.1.2, 8.1.3

[50] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012. 8.1.3

[51] E. Kasanen, K. Lukka, and A. Siitonen. The constructive approach in management accounting research. *Journal of management accounting research*, 5(1):243–264, 1993. 6.1

[52] D. E. Khelladi, R. Kretschmer, and A. Egyed. Detecting and exploring side effects when repairing model inconsistencies. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–126, 2019. 2.3.2, 8.2.1

[53] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez. The epsilon book. *Structure*, 178:1–10, 2010. 5.2.3

[54] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006. 5.2.3

[55] R. Kretschmer, D. E. Khelladi, and A. Egyed. An automated and instant discovery of concrete repairs for model inconsistencies. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 298–299. ACM, 2018. 8.2, 8.2.1

[56] M. Leonetti, L. Iocchi, and P. Stone. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence*, 241:103–130, 2016. 8.2.2

[57] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966. 2.3.2

[58] D. Lopes, S. Hammoudi, J. De Souza, and A. Bontempo. Metamodel matching: Experiments and comparison. In *2006 International Conference on Software Engineering Advances (ICSEA'06)*, pages 2–2. IEEE, 2006. 2.3.2

[59] J. J. López-Fernández, E. Guerra, and J. De Lara. Assessing the quality of metamodels. In *MoDeVVa@ MoDELS*, pages 3–12. Citeseer, 2014. 1, 2.3.1

[60] K. Lukka. The constructive research approach. *Case study research in logistics. Publications of the Turku School of Economics and Business Administration, Series B*, 1(2003):83–101, 2003. 6.1

[61] N. Macedo, T. Guimaraes, and A. Cunha. Model repair and transformation with echo. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 694–697. IEEE Press, 2013. 1, 8.1.3

[62] N. Macedo, T. Jorge, and A. Cunha. A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering*, 43(7):615–640, 2016. 1, 8.1, 8.1.2, 8.1.3, 8.2, A

[63] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015. 1

[64] F. Mantz, G. Taentzer, Y. Lamo, and U. Wolter. Co-evolving meta-models and their instance models: A formal approach based on graph transformation. *Science of Computer Programming*, 104:2–43, 2015. 8.2

[65] T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *International Conference on Model Driven Engineering Languages and Systems*, pages 200–214. Springer, 2006. 8.2

[66] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2018. 1, 3.1

[67] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276, 1999. 8.2.4

[68] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi. A survey on UML model smells detection techniques for software refactoring. *Journal of Software: Evolution and Process*, 31(3), 2019. 2.2.2

[69] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Complexity of finite-horizon Markov decision process problems. *Journal of the ACM (JACM)*, 47(4):681–720, 2000. 3.4

[70] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, et al. The relevance of model-driven engineering thirty years from now. In *International Conference on Model Driven Engineering Languages and Systems*, pages 183–200. Springer, 2014. 1

[71] G. Mussbacher, B. Combemale, J. Kienzle, S. Abrahão, H. Ali, N. Bencomo, M. Búr, L. Burgueño, G. Engels, P. Jeanjean, et al. Opportunities in intelligent modeling assistance. *Software and Systems Modeling*, 19(5):1045–1053, 2020. 7.2

[72] N. Nassar, H. Radke, and T. Arendt. Rule-based repair of EMF models: An automated interactive approach. In *International Conference on Theory and Practice of Model Transformations*, pages 171–181. Springer, 2017. 1, 8.1.1, 8.2

[73] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, A. Pierantonio, and L. Iovino. Automated classification of metamodel repositories: A machine learning approach. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 272–282. IEEE, 2019. 6.3, 7.1.3, 7.3.2

[74] M. Ohrndorf, C. Pietsch, U. Kelter, L. Grunske, and T. Kehrer. History-based model repair recommendations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–46, 2021. A

[75] M. Ohrndorf, C. Pietsch, U. Kelter, and T. Kehrer. ReVision: a tool for history-based model repair recommendations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 105–108. ACM, 2018. 1

[76] M. Ortega, M. Pérez, and T. Rojas. Construction of a systemic quality model for evaluating a software product. *Software Quality Journal*, 11(3):219–242, 2003. 5.2.3

[77] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010. 3.5

[78] J. P. Puissant. Resolving inconsistencies in model-driven engineering using automated planning. In *Seminar on Advanced Tools & Techniques for Software Evolution (SATToSE), Koblenz, Germany*, 2012. 8.1.3, 8.2.2

[79] J. P. Puissant, R. Van Der Straeten, and T. Mens. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, 14(1):461–481, 2015. 8.2.2

[80] A. Reder and A. Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 220–229, 2012. 8.2.1

[81] E. Ries. Minimum viable product: a guide. *Startup lessons learned*, 3, 2009. 6.1

[82] J. Rumbaugh, I. Jacobson, and G. Booch. The unified modeling language. *Reference manual*, 1999. 2.1

[83] J. Schoenboeck, A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Schwinger, M. Wimmer, and M. Wischenbart. CARE: a constraint-based approach for re-establishing conformance-relationships. In *Proceedings of the Tenth Asia-Pacific Conference on Conceptual Modelling-Volume 154*, pages 19–28, 2014. 8.1.1

[84] Seal-Ucla. SEAL-UCLA/Ref-Finder. Last accessed on 24/03/2021, `https://github.com/SEAL-UCLA/Ref-Finder`. 8.2.4

[85] S. Shafiq, A. Mashkoor, C. Mayr-Dorn, and A. Egyed. Machine learning for software engineering: A systematic mapping. *arXiv preprint arXiv:2005.13299*, 2020. 1, 8.2.3

[86] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014. 3, 3.2

[87] B. K. Sidhu, K. Singh, and N. Sharma. A machine learning approach to software model refactoring. *International Journal of Computers and Applications*, pages 1–12, 2020. 8.2.3

[88] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008. 2.1, 2.2.1, 2, 5.2.1, 5.2.1, 5.2.2

[89] E. Syriani, R. Bill, and M. Wimmer. Domain-specific model distance measures. *Journal of Object Technology*, 18(3), 2019. 1

[90] G. Taentzer, M. Ohrndorf, Y. Lamo, and A. Rutle. Change-preserving model repair. In *International Conference on Fundamental Approaches to Software Engineering*, pages 283–299. Springer, 2017. 1, 2.3.2

[91] S. Thrun and M. L. Littman. Reinforcement learning: an introduction. *AI Magazine*, 21(1):103–103, 2000. 3.3, 5, 3.3, 4.2, 7.1.3, 8.2.4

[92] D. Torre, Y. Labiche, M. Genero, and M. Elaasar. A systematic identification of consistency rules for UML diagrams. *Journal of Systems and Software*, 144:121–142, 2018. 2.2.3, A

[93] L. Torrey and J. Shavlik. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. IGI Global, 2010. 3.5

[94] J. Wust. Sdmetrics: The software design metrics tool for UML, 2005. 1, 2.3.1, 5.2.1, 5.2.3, 8.2.3, A

[95] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 315–324, 2009. 8.1.2

[96] Önder Babur. A labeled Ecore metamodel dataset for domain clustering, Mar. 2019. 7.1.5

Part II

ARTICLES

PAPER

IMPROVING MODEL REPAIR THROUGH EXPERIENCE SHARING

A. Barriga, A. Rutle, and R. Heldal.

In Journal of Object Technology, Volume 19, Number 2, 2020.

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Improving model repair through experience sharing

Angela Barriga[a] Adrian Rutle[a] Rogardt Heldal[a]

a. Western Norway University of Applied Sciences, Norway

Abstract In model-driven software engineering, models are used in all phases of the development process. These models may get broken due to various editions throughout their life-cycle. There are already approaches that provide an automatic repair of models, however, the same issues might not have the same solutions in all contexts due to different user preferences and business policies. Personalization would enhance the usability of automatic repairs in different contexts, and by reusing the experience from previous repairs we would avoid duplicated calculations when facing similar issues. By using reinforcement learning we have achieved the repair of broken models allowing both automation and personalization of results. In this paper, we propose transfer learning to reuse the experience learned from each model repair. We have validated our approach by repairing models using different sets of personalization preferences and studying how the repair time improved when reusing the experience from each repair.

Keywords Model Repair; Reinforcement Learning; Transfer Learning

1 Introduction

Models are often used to develop key parts of systems in engineering domains [WHR14]. In model-driven software engineering (MDSE) processes, models become more prone to errors as changes occur in their development environment, such as growing modeling teams or modifications in requirements. Tools that automate or support error detection and repair of models can improve how organizations deal with these errors. Model repair research has produced diverse tools that tackle repair of faulty models from different perspectives: e.g., support systems with abstract repairs [OPKK18], rule-based [NRA17] or automated approaches [MGC13]. Despite the variety of approaches, the proposed solutions can be arranged in two different lines of research: *support systems* where the repair choice is left to the developer's criteria or *fully automatic*, non-interactive model repair. Both approaches present advantages and disadvantages. Support systems that personalize the repairing process provide tailor-made solutions, however, they are time-consuming since they require close interaction from the modeler and are hard to scale for repairing a wider range of models. Automatic solutions improve

Angela Barriga, Adrian Rutle, Rogardt Heldal. *Improving model repair through experience sharing*.

In Journal of Object Technology, vol. 19, no. 2, 2020, pages 13:1–21. doi:10.5381/jot.2020.19.2.a13

repair time, however, they have the drawback of providing the same solutions for the same errors although different modelers may have different preferences for repairing the same model. A desirable solution should provide a balance between automation and personalization of repair [MJC16], facilitating the use of both approaches' advantages.

This paper follows our previous work [BRH18, BRH19], where we proposed reinforcement learning (RL) [SB11] as a solution to allow both automatic and personalized model repair. RL consists of algorithms able to learn by themselves how to interact in an environment only needing a set of available actions and rewards for each of these actions. The structure of RL algorithms provides the necessary flexibility to adapt to different personalization settings and to perform faster after each execution. Following this approach, we implemented our tool PARMOREL (Personalized and Automatic Repair of MOdels using REinforcement Learning) [Bar] where users can personalize the repairing process. By utilizing RL, the repair gets faster since PARMOREL learns from the errors which have been already faced. We validated the tool's usefulness by repairing randomly generated models under one set of user preferences [BRH19].

In this paper, we focus on repairing and learning from different sets of preferences by applying transfer learning (TL) to reuse the experience gained from repairing under different personalization settings. TL is a research line in machine learning (ML) that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem to solve it faster. TL permits us to share and reuse the experience gained in different users' repairs. Our objective is to improve the repairing time by avoiding repeated calculations for errors to which a solution is already learned. The contributions of this paper are hence (i) the application of RL to produce personalized model repair solutions, (ii) an approach to improve model repair time with TL, and (iii) a proof of concept implementation.

The remainder of this paper is structured as follows. Section 2 provides a motivating example for our approach. Section 3 explains the necessary background of PARMOREL and RL to understand the rest of the paper. In Section 4, we explain our approach of TL for model repair. Next, Section 5 presents the implementation and testing of our approach through two different examples. After discussing the threats to validity in Section 6 and related work in Section 7, we conclude the paper and present future work plans in Section 8.

2 Motivation

Model repair is a broad field that covers different model issues: syntactic and semantic errors, design smells, compliance to quality attributes and metrics, co-evolution issues, etc. We focus on developing a framework that simplifies how modelers repair and improve their models regardless of the model's type, the type of issues they repair, and the user's expertise. To do so, we utilize ML algorithms which provide enough flexibility to handle the above-mentioned variety of models and issues.

When following rule-based or quick-fix approaches, the modeler must specify a series of rules to repair issues in the model or derive them from grammar or constraints. Although these rules are precise for a single set of requirements they are not universal and might not satisfy every specific modeler's requirements. The number of rules to define can increase rapidly when repairing big models. In contrast, ML algorithms are easy to scale, as they can target any model size without increasing the number of rules. This is due to the learning capability of ML algorithms which allows them to learn how to repair without being explicitly programmed for every situation.

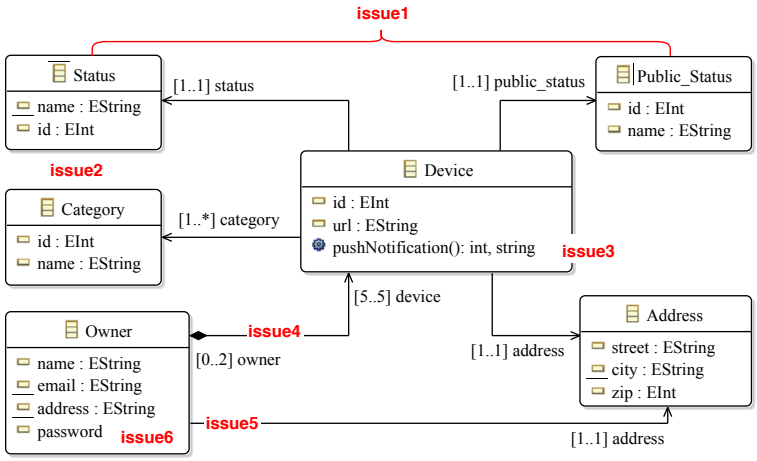


Figure 1 – Sample model containing a variety of issues

In our previous work [BRH19], we used RL in our framework and applied it to repair syntactic errors in models that violated certain constraints of the Ecore metamodel [SBMP08]. In the current work, we address a wider range of issues, including design smells and compliance to quality properties [BV10, BDRIP19]. Furthermore, we take the learning capability of our framework one step further, by utilizing TL, which enables experience sharing between different users.

Consider as an example the model shown in Fig. 1, which represents a part of a smart system in which a device has several statuses, categories, owners and an address. The model contains several types of issues: design smells, i.e. classes with duplicated attributes and references (issues 1 and 2), syntactic inconsistencies corresponding to constraints imposed by the language used to define the model, i.e. the Ecore metamodel [Fou] (issues 3, 4 and 6: operation with two return parameters instead of a single one, containment reference with an upper bound greater than 1 and attribute without a type) and violations with respect to some quality properties, i.e. attributes should not be (potential) associations [LFGDL14] (issue 5).

Each of these issues can be addressed by applying different actions. For example, issue1 could be handled by deleting or updating one of the duplicated classes Status or Public_Status or by creating a hierarchy within these classes. This hierarchy could consist of promoting one of the initial classes to superclass or making both of them children of a new superclass. References could remain in the children or belong to the superclass. Likewise, issue4 could be repaired by (i) changing the upper bound from 2 to 1, (ii) modifying the containment to a regular reference, (iii) deleting the reference owner or (iv) deleting both owner and device. issue5 by renaming or removing address in Device, class Address or the address association and issue6 by setting a type or deleting the faulty attribute or the container class.

This sample model shows that addressing model issues is not a trivial task. There are multiple, possible repair solutions that a modeler could choose while there might not exist an objectively best solution to satisfy all modelers. ML algorithms can provide model repair solutions adapted to each modeler without requiring to specify beforehand how to act for every specific model and modeler requirement.

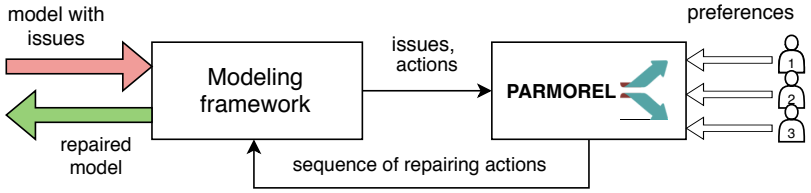


Figure 2 – Overview of our approach

3 Reinforcement Learning in PARMOREL

This section introduces a brief notion of RL and PARMOREL in order to provide a comprehensive guide to understand the rest of this paper. Figure 2 presents an overview of our approach. PARMOREL uses RL to find a sequence of concrete actions required to repair the issues present in a model. We rely on an external modeling framework (i.e. the Eclipse Modeling Framework (EMF) [Fou, SBMP08]) to retrieve issues in the models (e.g. attribute without a type, duplicated class). The modeling framework is also responsible for applying the actions selected by PARMOREL (e.g. setType, delete or addSuperClass) and creating the repaired models. PARMOREL produces the sequence of actions to repair each model based on preferences introduced by the user. At the moment, users can select preferences from a catalogue of predefined options offered through a GUI (see Fig. 6).

In the beginning, RL follows a try-and-fail approach: when a sequence of concrete action applications leads to a repair which is aligned with the user preferences, the actions in that sequence will get rewarded, and otherwise punished (negative reward when punishing, positive when rewarding). Following this approach, the algorithm learns which actions to apply for each issue. We adapt RL rewards to align with user preferences; e.g., if a user prefers to preserve the structure of the original model, we can assign positive rewards to conservative actions. Moreover, we filter actions obtained from the modeling framework so that PARMOREL only works with those that can be applied in a concrete type of error (see line 5 in Alg. 1); e.g., if one issue is present in an attribute, actions invokable in references or classes would be discarded. This way, we improve performance by reducing the search space.

RL provides structures to store experience gained from each repair, so that the algorithm can improve its performance in consecutive executions. PARMOREL is powered by the Q-learning algorithm [SB11], in which experience is stored in a table structure called *Q-table* (see example in Fig. 3). We chose Q-learning because it provides several features that are useful to solve the model repair problem: (i) the Q-table structure allows us to pair actions with issues and locations in the models (see below), (ii) the Q-table is highly reusable and easy to import and export into new executions, (iii) the algorithm is able to find a set of different solutions for the same issue thanks to its combination of exploitative and explorative policies (i.e., picking the best-known action vs. finding a new random one) and (iv) it takes into account the consequences of applying an action to measure its suitability. Traditionally, the Q-table stores pairs of states (a situation to solve) and actions. An action can be any editing operation that can be applied to the model and repair an issue, including element creation and deletion.

	e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
entry1 := issue5, class1: Owner, action1: delete	0	0	-130	-130	-130	-130	-130	-130	-260	-260	-260
entry2 := issue5, attrib1: address, action1: delete	0	0	0	73	73	73	73	73	73	73	73
entry3 := issue5, attrib1: address, action2: setName	0	0	0	0	0	0	93	196	196	289	**382**
entry4 := issue5, class2: Address, action1: delete	0	0	0	0	23	23	23	23	23	23	23
entry5 := issue6, class1: Owner, action1: delete	0	-130	-130	-130	-130	-260	-260	-260	-260	-260	-260
entry6 := issue6, attrib1: password, action1: delete	0	0	0	0	73	73	73	73	73	73	73
entry7 := issue6, attrib1: password, action3: setType	0	0	0	93	93	93	196	289	289	382	**475**

Figure 3 – Detail of how the Q-table gets populated each episode

In PARMOREL, we use a 3-dimensional Q-table to store *entries*, which corresponds with a combination of a concrete action applied in a location to repair a particular issue in the model. Each entry has a weight which reflects how good an action is for repairing an issue in a model location according to the user preferences. Actions from entries in the Q-table are stored individually and they are sequence-independent. To obtain these repairing actions, the algorithm filters the selected invokable actions to keep only those that are able to repair at least one error (see lines 6-9 in Alg. 1). Hence, the Q-table only contains entries that are able to repair an issue. Although these actions may produce different repairs depending on their application order, it is not necessary to store the whole sequence since the weights will be updated based on how good an action is both individually and in the applied repair sequence. For example, when repairing the model in Fig. 1, one entry in the Q-table would be: entry3 := issue5, attribute1: address, action2: setName with a final weight of 382 after 10 episodes (see below).

Figure 3 shows how the Q-table would be populated with weights in each episode of the algorithm when repairing issue5 and issue6 from Fig. 1. Each episode is one iteration in which the algorithm has successfully repaired the model within a predefined number of steps (see lines 10-18 in Alg. 1); one step corresponds to the application of one entry. After each episode, the algorithm starts repairing the model again in order to find possibly "better" repair sequences; i.e., sequences with entries whose total weights are higher than the currently found ones. Figure 3 represents 10 episodes (e0-e10); in the beginning (e0), the Q-table is empty as the algorithm does not know yet how to repair the model, hence all entries have a weight of 0. Picking the "right" number of episodes assures that the algorithm has enough time to find different possible sequences to repair the model; what is *right* depends on the model size, the number of errors and actions available. There is no established policy of how many episodes are best for a given problem [SB11], so according to our experimentation, between 15 and 20 episodes are enough. Likewise, the maximum number of steps is picked based on the size of the model and number of errors - 10 in the case of this example. Limiting the number of steps assures that the algorithm does not fall in an infinite loop while looking for repairing sequences. In the future, the selection of these values could be automated by implementing a hyperparameter selection method, similar to grid and random search in other ML algorithms [BBBK11].

For this example, we simulate a user who prefers to repair the model preserving as much as possible of its original structure (*pref1*) and to address each issue individually, rewarding entries that repair one issue at a time (*pref2*). His intention is to respect the original model and to repair in a way that allows him to track independently the repair of each issue. We would like to remark that the preferences displayed in

this paper work as a proof of concept to evaluate different repair scenarios with our approach and they may not substitute the requirements specified by a real modeler.

When an action fulfills pref1, its entry obtains a reward of 10, otherwise, it obtains a punishment of -10 multiplied by the number of times the entry is unaligned with the preference. For pref2, entries obtain a positive reward with the percentage of remaining non-repaired issues after repairing a single issue or a negative reward with the percentage of repaired issues. For example in e1, applying delete in class Owner would get *-80* from pref1 since it would delete 8 elements of the model (1 class, 4 attributes and 3 references) and *-50* from pref2, since we would be repairing 50% of the issues simultaneously (3 out of 6)—making a total reward of *-130*. In e3, however, changing the type of password would get a reward of 10 from pref1, since it solves the issue by updating an existing element without deleting or adding new ones to the model, and 83 from pref2 since it repairs just 1 out of 6 issues (13%) and does not interfere with the resting 83%—making a total reward of *93*.

If no preference is involved with the applied entry, there will be a positive default reward for each repaired error. Picking the right rewards for the preferences is done based on our experimentation, there is no established policy about defining rewards in Q-learning [SB11]. For simplicity, we calculate the weights in the Q-table with an accumulation of the rewards obtained from the user preferences. In the implementation in PARMOREL, however, these weights are calculated based on the Bellman Equation [Bel13] where one of the inputs to the equation is the rewards obtained from user preferences.

The Q-learning algorithm picks the entry with the highest value in the Q-table or an entry randomly (see line 13 in Alg. 1). This way, it assures applying new entries that would have otherwise never been picked; in each step, there is 20% chance of picking an entry which is not having the highest weight in the Q-table. Following this procedure, Fig. 3 displays how the algorithm picks entries in each episode: highest in green, random in blue. Once a weight is stored, it is propagated to the following episodes and if the entry is picked in multiple episodes the weight will be accumulated. This way, after some episodes the algorithm is able to learn which are the entries most aligned with user preferences (see lines 23-24 in Alg. 1); for the current user these entries are entry3 and entry7. Below, we show a pseudo-code of Q-learning within PARMOREL (see Alg. 1).

Due to the Q-table's storage procedure, when facing the same error repeated times, even if it appears in different and unrelated models, PARMOREL will be able to recognize it and gradually repair it in a more efficient way. For example, issue5 in Fig. 1 will always follow the same structure —an attribute with the same name as class/association—regardless of the model where issue5 appears.

In traditional RL, the weight of each entry depends on a single reward ; e.g., for a robot learning how to escape a maze, it receives a negative reward when stepping into a wall and a positive one when entering a free space. However, in model repair one entry's weight may depend on multiple rewards since it might involve several user preferences, e.g., recall that entry1 in Fig. 3 got its weight based on two different preferences. Introducing user preferences complicates reusing experience since what is a good repair for one user might not be acceptable for another one. This challenge of reusing experience when the rewards change from one scenario to another is addressed in the ML field by TL [PY10].

Algorithm 1 Q-learning in PARMOREL

```
1: INPUT: from User (model, preferences)
2: INPUT: from M. framework (issues ← getIssues(model), actions ← getActions(model),
3: locations ← getLocations(issues))
4: for each i in issues do
5:     invokableActions ← getAllInvokableActions(i)
6:     repairingActions ← getAllRepairingActions(i, invokableActions)
7:     for each a in repairingActions do
8:         addQtableEntry(i, a, i.location, 0)
9: originalModel ← model
10: while numberOfEpisodes not 0 do
11:     while numberOfSteps not 0 ∨ issues != ∅ do
12:         e ← selectRandomIssue(model)
13:         entry ← selectEntry(e, Q-table) // random or highest Q-value
14:         model.applyAction(entry.getAction(), entry.getLocation)()
15:         rewards ← getRewards(model, preferences)
16:         updateQtableWeights(entry, rewards)
17:         seq ← addEntry(entry)
18:         if checkNewIssues(model) then repeat lines 4-9
19:     sequences ← addSequence(seq)
20:     numberOfEpisodes ← numberOfEpisodes - 1
21:     model ← originalModel
22: bestSequence ← getBestSequence(sequences)
23: updateQtableWeights(bestSequence.getEntries, rewards)
24: applySequence(bestSequence, model)
25: OUTPUT: repaired version of model
```

4 Applying transfer learning in model repair

TL differs from traditional ML in the fact that, instead of learning how to solve a problem from zero, it reuses experience gained in solving a source task to accelerate the solution of a new target task. The benefits of TL are that it can speed up the time it takes to develop and train an ML system by reusing already developed solutions.

There exist many techniques within TL. In PARMOREL we take into account starting-point and imitation methods [TS10]. Starting-point methods use the solution found in the source task to set the initial experience in a target task. Imitation methods use parts of the source task experience to influence the solution of the target task. Applied to our scenario, following starting-point methods the whole Q-table from a previous repair would be reused in a new one while following imitation methods only some parts of the source Q-table would be copied to the new repair.

4.1 Learning through propagating preferences

As mentioned, while repairing models with PARMOREL the Q-learning algorithm stores weights in the Q-table indicating how good an entry is for repairing an issue. Working with the same Q-table in different repair scenarios is useful as long as user preferences remain unchanged. However, it is not convenient to directly reuse the Q-table (as in starting-point methods) when introducing new sets of preferences since the repairing process would use the weights calculated with the old preferences and this could lead to repair decisions unaligned with the new ones. Following imitation methods would not be convenient either since we would still copy some of the weights from an old Q-table calculated with old preferences. Our goal is to reuse the experience obtained from other users' repairs, therefore we apply our own version of the starting-

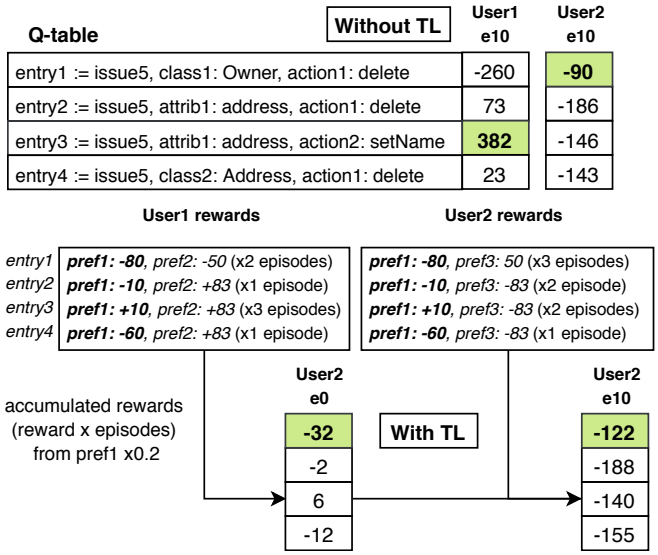


Figure 4 – Example of differences when initializing the Q-table with and without TL

point method by copying all Q-table entries without their weights so that the algorithm would not start with a completely empty Q-table. In addition, we apply a variant of the imitation method in which instead of copying weights from the Q-table, we keep track of which preferences were used to produce the weights during the episodes, accumulate their values, and reuse those which are aligned with the new user preferences.

The quality of an entry is no longer tied to a specific set of preferences; the algorithm is now able to pick the individual rewards used to calculate the weight of each entry. Since entries represent issues and actions that can potentially appear in any model, the structure of the Q-table can be reused regardless of the model to repair.

Note that the sample Q-tables shown in this paper exemplify the entries by displaying the names of the locations where the actions are applied; e.g., entry3 in the Q-table in PARMOREL would look like entry3 := issue5, attribute1, action2 rather than entry3 := issue5, attribute1: address, action2: setName.

As an example, the upper part of Fig. 4 shows the difference in the Q-table of two users with different preferences for repairing issue5 in Fig. 1. User1 is the same user we simulated in Fig. 3 with pref1 and pref2 as displayed in Fig. 4. User2 shares pref1 and in addition prefers to repair as many issues as possible with just one action (pref3).

These users share one of their preferences, but since the other one is different, they will get different repairs. The Q-table reflects this difference, entry3 is the one selected for User1, and entry1 for User2, since by deleting the class Owner we repair issue4, issue5 and issue6 at the same time. User2 preferences are specially interesting because it shows how RL is able to pick a solution when preferences are contradictory. In this situation, it is not possible to repair more than one error at a time without deleting several elements in the model, which goes against pref1.

Additionally, the lower part of Fig. 4 shows how the weights of User2 are changed when we transfer learning. That is, by transferring the accumulated rewards (multiplied by the number of episodes they were applied) coming from shared preferences between both users (pref1 in this example), it is possible to streamline consecutive repairs.

When sharing experience, we initialize the Q-table with the accumulated rewards of the shared preferences multiplied by a discount factor of 0.2. This way we assure previous repairing processes influence the new repairs by jump-starting the repairing process but do not interfere with learning new repair sequences. Based on our experimental results, we found that a value of 0.2 gave the best results for our cases. This parameter's value can be modified so that the previous experience affects less or more new repairs. However, the value should remain a constant during the execution otherwise some parts of the experience will be more favoured than others. Now, when User2 starts repairing, PARMOREL will already know that entry3 is the best according to pref1, but thanks to the discount factor it is still able to find a better solution for this user.

4.2 Integration with PARMOREL

In this section, we detail how PARMOREL shares experience between different users. We use the model in Fig. 5 to illustrate how PARMOREL supports TL in model repair. The learning information gained after each repair is represented by the concept Experience which is composed of one to many entries and preferences. Experience has a structure that achieves transfer learning from older repairs independent of the models which we want to repair.

The concept Entry has references to all the elements that are part of the Q-table: an Issue, a Location and an Action. In addition, an Entry has a zero to many references to Reward. Weights are not included in this model since we only share the accumulated rewards which were used to calculate them. Hence, the Reward contains a numerical value based on the users' preferences. The model has also the following constraints:

1. There cannot exist two identical Entry elements (formed by the same combination of Issue, Location and Action).

2. One Entry cannot contain more than one Reward from the same Preference.

When the repairing algorithm is executed for the first time, there is no previous experience and PARMOREL starts learning from zero; the Q-table is empty and the algorithm needs to process the model to populate it with entries. When the repair finishes, the first experience is created. It contains every entry stored in the Q-table and the accumulated rewards coming from user preferences for those entries. For second and consecutive executions, the sharing will be different depending on how much current user preferences coincide with the ones already stored in the Experience:

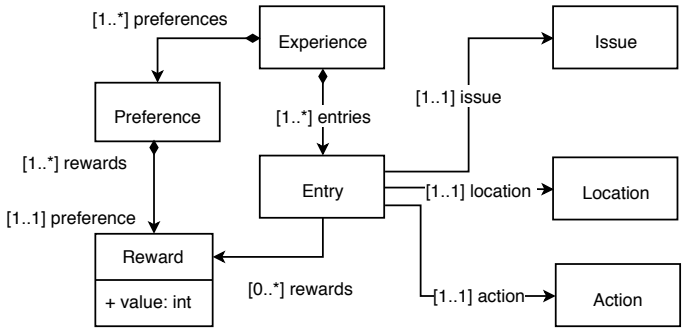


Figure 5 – Model of transferring learning experience in PARMOREL

- Always, independently of the new preferences chosen, the current Q-table is initialized with the stored entries (see line 6 in Alg. 2). If there are new entries in the current model, these are also added to the Q-table and therefore in the experience (lines 4-8 in Alg. 1).
- If any of the new preferences are shared with the stored ones, the current Q-table is initialized with all rewards coming from the matching preferences (see lines 7-10 in Alg 2). Returning to the example in Fig. 4, User2's Q-table is initialized with rewards correspondent with pref1 since that is the one selected by both users. These preferences' rewards will be updated for future propagation (see line 12 in Alg. 2). If a user introduces preferences not stored yet, these will be added to the experience.

When sharing experience in PARMOREL, we reduce the random factor of the Q-learning algorithm from 20% to 10% to enhance the influence of the previous Experience. The number of episodes is also reduced since due to TL the repairing process is improved and solutions are found faster.

Regarding Alg. 1, for introducing TL, we add some new code right after the inputs for checking if any previous Experience exists to initialize the Q-table, see Alg. 2. Lines 4-9 in Alg. 1 are executed for those errors not present in the experience. Then, after line 16 we store pairs of preference-reward for the selected entry, in order to keep track of which preferences provided each reward. When facing the same entry, the pairs are updated, accumulating the rewards. Finally, at the end of the algorithm, we store all generated experience in a text file with XML format (see output in Alg. 2).

Algorithm 2 Transfer learning in PARMOREL

```
1: INPUT: from User (preferences)
2: INPUT: from PARMOREL (experience, discountFactor, episodes)
3: Qtable ← createNewQtable()
4: if experience != ∅ then
5:     reduceNumberofEpisodes(episodes)
6:     for each entry in experience.entries do
7:         addQtableEntry(entry, 0)
8:     for each pref in preferences do
9:         for each reward in entry.rewards do
10:            if reward.preference == pref then
11:                updateQtableWeight(entry, reward.value * discountFactor)
12: //Algorithm 1, in line 4: if i exists in the Qtable then skip loop
13: //after line 16: for each entry, store in the experience reward values coming from preferences
14: updateExperience(Qtable.entries, preferences, rewards)
15: OUTPUT: XML with generated experience
```

5 Implementation and Evaluation

In this section, we present a proof of concept implementation of our approach, testing it with two examples: we repair a broken model with different sets of preferences and then we repair 30 randomly mutated models obtained from 3 originals from GitHub. The objectives of this section are to show that our approach can (i) store and reuse experience learned from different preferences and (ii) improve the repairing time when working with different models. The implementation source code and the models are publicly available in [Bar]. Additionally, PARMOREL is available as an Eclipse plugin (see Fig. 6).

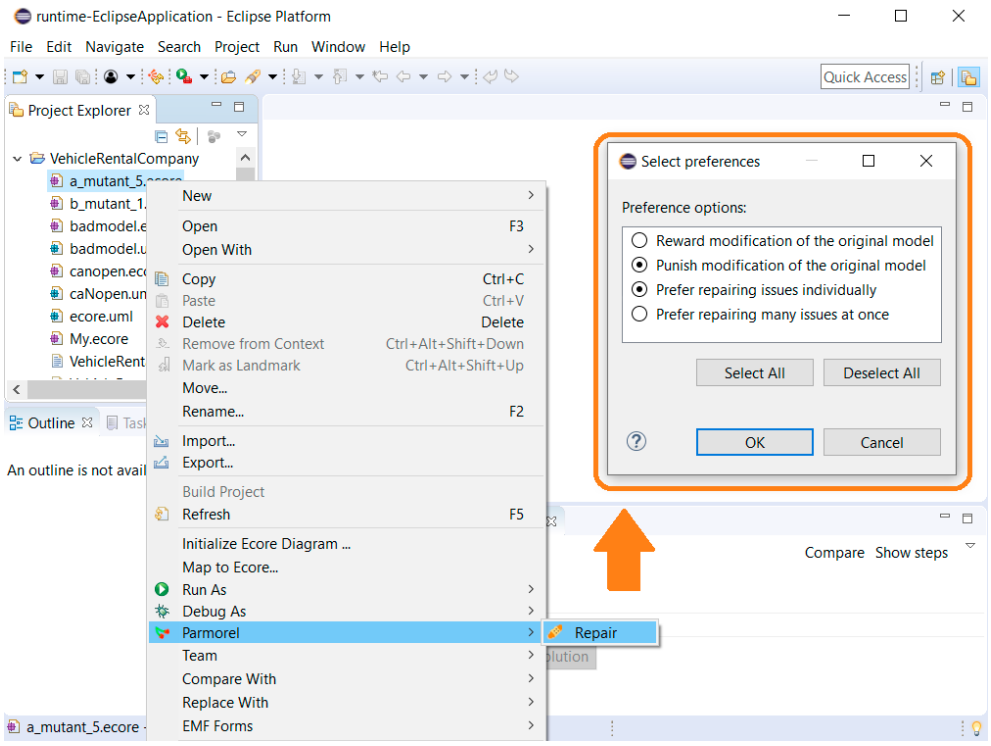


Figure 6 – Screenshot of PARMOREL Eclipse plugin

In our current PARMOREL implementation, we use the EMF API to obtain issues present in the model and actions to repair it. For these examples, PARMOREL is run in Eclipse Oxygen (the Modeling package) on a laptop with the following specifications: Windows 10 Home, Intel Core i5-6300U @2.4GHz, 64 bits, 16GB RAM.

5.1 Example I: different users repairing the same model

In this example, we use our implementation of TL in PARMOREL to repair the broken model presented in Section 2 (see Fig. 1). We simulate 7 different users with different sets of preferences to repair the model. For the sake of brevity, we only display the first three users in Fig. 7 together with their preferences and the repaired model that each of them obtains. Our goal with this example is to demonstrate that our approach is able to produce different repair solutions depending on the preferences selected by the user and to streamline the repairs the more experience is reused.

Each user preferences are a combination of those displayed in Fig. 6. The combinations have been chosen so that they are completely different (User1 and User2 in Fig. 7), coincide partially (User2 and User3), and are the same. Also, some users may completely coincide in some of their preferences while having opposite preferences in others (User3 and User5 coincide in repair errors individually but one prefers to preserve the original model and the other to modify it). This diverse set of preferences allows us to evaluate if our approach is able to: (i) share experience between users with unrelated preferences, (ii) successfully reuse experience when preferences coincide completely or partially with the stored experience, and (iii) achieve better performance when more parts of the experience are reused.

The first repair of our example is executed with User1's preferences (see Fig. 7), when there is no previous experience stored. Afterwards, we repair the model using each set of user preferences in numerical order. The experience gained is stored and reused in the next repair. Note that repair processes are not concurrent but sequential: when one user finishes his process the experience is locked until the next user starts repairing. We also changed the repair order but it did not provide different solutions.

Figure 7 shows, for each user, the applied repairing sequence and the repaired model which PARMOREL produces. In the repaired models, we show where each issue was repaired. Below, we detail the repairing process for each user in Fig. 7 :

- **User1**: Since this is the first repair, there is no experience stored yet. In order to preserve the original model, PARMOREL avoids to delete or add elements to the model as much as possible.
- **User2**: This repair reuses experience obtained from the previous user's repair process, however, since User2's preferences do not coincide with the ones stored (User1's preferences), only entries without rewards are reused. User1's preference is opposite to User2's (preservation vs modification of the original model), therefore, this repair is not influenced by User1's preference. Since User2 wants to reward modification, the algorithm chooses to delete elements in the model and to add a new class to solve issue1 and 2.
- **User3**: This user is the first one to pick preferences already stored in the experience, in addition to a new one. As in the previous repair, one of the preferences selected by User3 is new. Since User3 prefers to reward model modification and to repair errors individually, there are fewer elements deleted than in the previous repair.

The random component of RL produces variations in the results, therefore, to get stable results, we reproduced the following steps 20 times: repairing the broken model with preferences from User1 to User7 starting with no previous experience in User1. Results in Fig. 7 are the majority of those that were obtained most frequently.

To check if our approach succeeded in improving the repair time when more experience is reused, we measured the time used to complete each user's repairing process. Figure 8 shows how long it takes to repair the model with the 7 sets of user's preferences during three different rounds. Although there are some variations in each round, we can see a pattern. User1's execution had no previous experience, therefore this repair takes longer, where the preprocessing of entries took an average of 570ms. User2 shows a faster execution than before since they reuse entries and PARMOREL does not need to calculate them again. From here, we can see how execution time gets even lower since User3 to User7 have preferences that appear in the stored experience. User3 introduced new preferences so the repair is not fast as in User4 to User7 since all their preferences were already stored in the experience. In these last users, their execution times are not so different. This is because the type of preferences introduced also influences execution time, e.g., repairing several issues at a time is faster than individual repair.

In conclusion, each user obtains a customized repairing process and a repaired Ecore file is exported. PARMOREL allows to automatically store and share experience in different executions. Sharing is adapted depending on whether users introduce preferences already stored in the experience (reuse of entries and rewards) or not (reuse of only entries). With this approach, the repairing time becomes faster when reusing more experience.

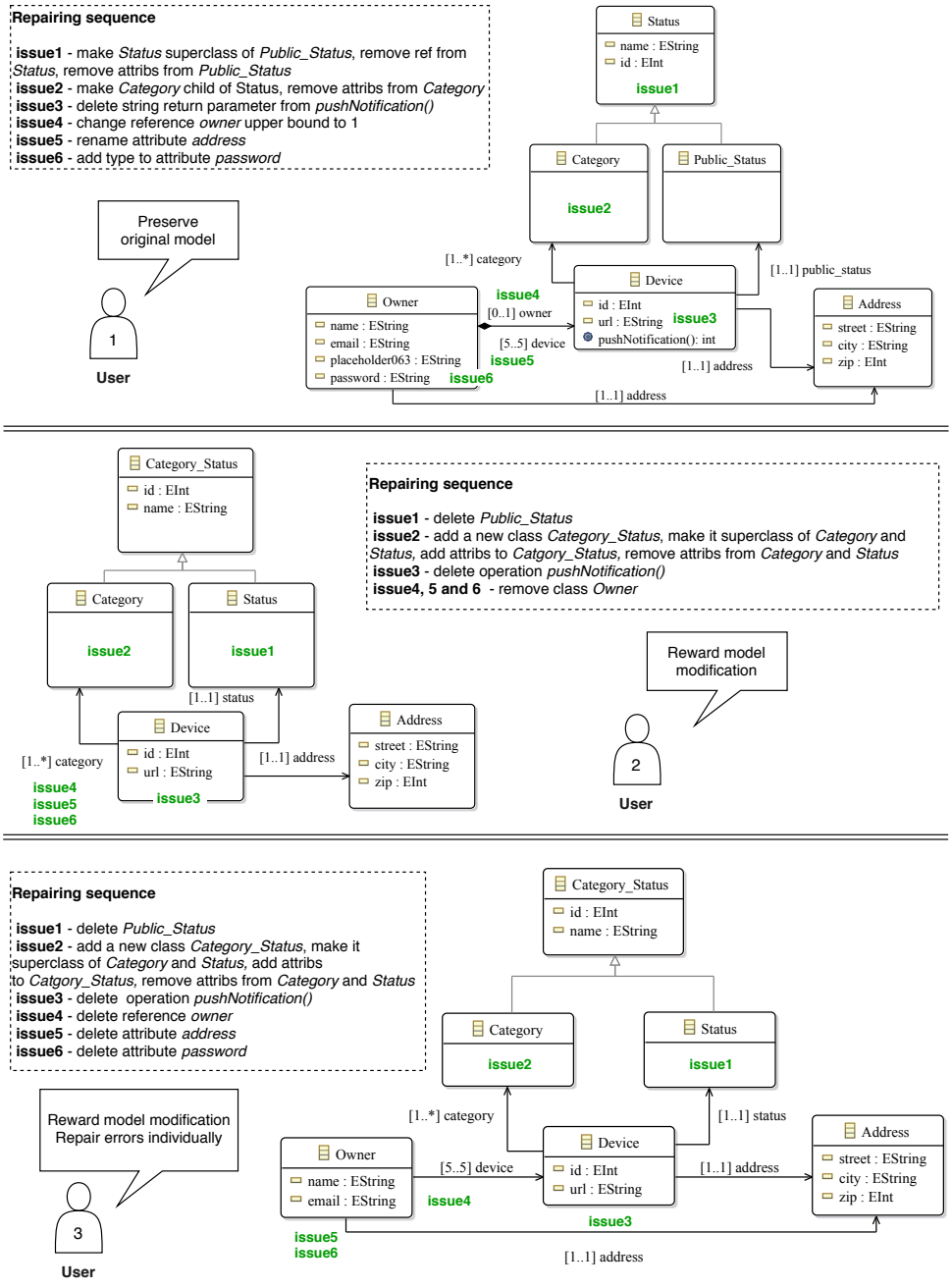


Figure 7 – Users with different preferences repair the same broken model

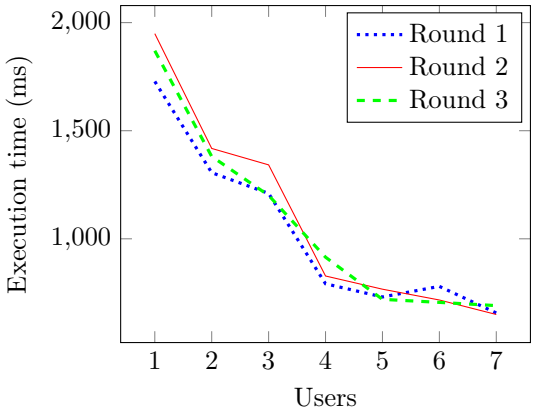


Figure 8 – Evolution of repair times for model in Fig. 1 with 7 users in 3 different rounds

5.2 Example II: different users repairing randomly mutated models

To evaluate and test the generality and scalability of our approach, we use our implementation of TL in PARMOREL to repair 30 mutant models generated from 3 industry size models (10 mutants per original model) obtained from GitHub: RandomEMF [mar15], OCCIware ecore [Occ17] and amlMetaModel [aml16]. To generate the mutants, we use AMOR Ecore Mutator [AKK+08], an EMF-based framework to randomly mutate models conforming to the Ecore meta-metamodel. We refer to each group of mutants coming from these models as batches A, B, and C, respectively. AMOR offers different mutation options such as deleting, adding, or moving objects, however, here we only introduce mutations by updating features of the original models. This is because updating features would create issues which are similar enough to demonstrate reuse of experience (in our previous work [BRH19], we demonstrated how PARMOREL could repair a more diverse variety of mutations through 100 models). The mutant models contain between 1 and 7 syntactic errors (out of 11 different issues) and have different sizes ranging from 21 to 36 classes, 21 to 100 attributes and 15 to 197 references. Actions to repair the mutants are directly extracted from the editing actions available in EMF. Mutants from this example are available to download in [Bar]. In this example, we simulate 3 users (User1 to User3) to repair each batch of mutants (each user repairs one batch) with different set preferences based on those displayed in Fig. 6.

First, we repair each batch without applying TL. Then, we proceed to repair them in the following order: A-B-C-A, starting from zero with no experience stored in PARMOREL. We repair batch A twice to measure its repairing time with and without TL, since in its first repair there is no experience to propagate. Results are displayed in Table 1 (times displayed are the average after reproducing this cycle 20 times). In batch A, the repairing time is improved 58,90% when using TL; it benefits from the experience containing all its errors and preferences since we repair A twice in the cycle. Batch B contains 10 errors, from which 3 are not present in the experience and User2's preference is new, so it only gets an improvement of 9,71%. Finally, for mutants in batch C, we obtain an improvement of 36,56%, a better result than the previous batch since PARMOREL does not face any unknown issue and has already processed one of User3's preferences. Batch C takes longer to repair with and without TL since it contains the biggest models.

The results of this evaluation indicate that our TL approach accomplishes sharing the experience learnt by repairing different models, can work with real-world models and streamlines the repair regardless of the chosen preferences.

	Per model		Total batch		
	Elements	**Issues**	**Without TL**	**With TL**	**Improvement**
Batch A	76	1 - 7	7,93s	3,26s	58,90%
Batch B	74	1 - 5	7,41s	6,69s	9,71%
Batch C	336	1 - 6	12,06s	7,65s	36,56%

Table 1 – Comparison of repairing times with and without TL

6 Threats to validity

Although we consider our approach successful in integrating TL in PARMOREL and streamline the repairing process, we face some validation issues worth discussing in this section.

Models and errors. Our evaluation focused on repairing a broken model designed by us (see Fig. 1) and 30 mutant models produced with AMOR. The criteria for selecting which issues should be present in the first model was to create a broken model that could be repaired in different ways according to our set of preferences, making a motivating example for our approach. The reasons for using AMOR are its easy integration with EMF and the randomness of the introduced mutations. Despite this randomness, it has a predefined set of mutations, and the issues it produces might not be as complex as errors introduced by a human. Still, we believe it is realistic to think these issues could appear in real modeling environments.

Preferences. Since no real users participated during the evaluation, we simulated different sets of high-level repairing preferences. Although these preferences were fictitious, we consider them a good example for showing the potential of RL to produce personalized solutions and how experience sharing works.

Generality. Our approach is evaluated using EMF and Ecore metamodels, however, PARMOREL should also work with other types of models. The assumption is, as long as the framework in which the models are defined can detect errors and provide an API for editing actions, future versions of PARMOREL should be able to repair them and to apply TL.

Dynamic learning. Although we consider our approach to provide a balance between automation and personalization, it is obvious that providing a predefined set of preferences might not be universally applicable in all scenarios. Therefore, we contemplate the possibility of providing further interactions with users, for example, offering runtime repair options to the users and learning from their choices.

7 Related work

Model repair is a research field that has drawn the interest of many researchers to formulate approaches and build tools to repair broken models. Even though some of these tools offer some degree of automation and customization, we could not find in the literature any research applying RL or TL to model repair.

One tool that allows customization is Echo [MGC13], in which users can customize repair operations. They provide concrete repairs and produce well-formed model instances. The only output is the generated instance of the model, so the user lacks information about repair plans and causes of the inconsistencies. It has some predefined metrics such as preferring least-change options, which cannot be modified by users.

Taentzer et al. [TOLR17] present a prototype based on graph transformation theory for change-preserving model repair. In this approach, the authors check operations performed on a model to identify which ones caused inconsistencies and apply the correspondent consistency-preserving operations, maintaining already performed changes on the model. Although this approach does not offer active customization, it keeps track of user history and takes their repairing preferences into account. By obtaining preferences from historical data, this approach assumes user preferences will not change from one repair to another, which is a situation that could happen frequently when facing different model repair scenarios.

Other efforts focus on interactive solutions, authors in [CvBLR+17] present an interactive repairing tool powered by visual comparison of models, performing conformance checking. They claim fully automated methods lead to overgeneralized solutions that are not always adequate, and strong interaction comes with a high computational effort, therefore as future work they seek an equilibrium between automation and interaction. This is exactly our vision: a balance between the algorithm independence and enough interaction with the user to provide personalized solutions.

Khelladi et al. [KKE19] present a model repair approach that ranks repairs depending on the positive or negative side effect they produce. They also identify alternative repair paths and cycles of repairs. This is a very interesting research line and some of their concepts are also present in PARMOREL's implementation. We also avoid falling in cycles of repairs by delimiting the number of steps in the Q-learning algorithm, repairs with bad side effects will get a poor reward and the random component of Q-learning lets us explore different alternative repair paths. As future work, it would be interesting to integrate their concept of positive side effect to provide good rewards in PARMOREL.

Kretschmer et al. introduce in [KKE18] an approach for discovering and validating values for repairing inconsistencies automatically. Values are found by using a validation tree to reduce the state space size. Trees tend to lead to the same solutions once and again due to their exploitation nature (probing a limited region of the search space). Differently, RL algorithms include both exploitation and exploration (randomly exploring a much larger portion of the search space with the hope of finding other promising solutions that would not be selected normally), allowing to find new and, sometimes more optimal solutions for a given problem.

Also tree-powered, Model/Analyzer [RE12] is a tool that, by using the syntactic structure of constraints, determines which specific parts of a model must be checked and repaired. The user is expected to select a specific violation to be repaired but does not support user customization.

Puissant et al. propose a tool called Badger based on an artificial intelligence technique called automated planning [PVDSM15]. Badger generates sequences that

lead from an initial state to a defined goal. It has a set of repaired operations to which users can assign costs and weights to decide its priority. Badger generates a set of plans, each plan being a possible way to repair one error. We prefer to generate alternative sequences to repair the whole model since some repair actions can modify the model drastically. This makes it difficult for the user to decide which action to apply without knowing how it affects the rest of the model. Additionally, in Badger users can personalize parameters of a predefined set of operations, we offer higher-level preferences that allow a wider range of customization. Also, by combining RL and TL we are able to streamline the repairing process; while automated planning performance does not improve with time.

Lastly, it is worth mentioning search-based and genetic algorithm-based approaches since, although they have not been applied yet to model repair, they can be considered as possible competitors to RL. These techniques have shown promising results dealing with model transformations and evolution scenarios, for example in [KMW+17] Kessentini et al. use a search-based algorithm for model change detection. These algorithms deal efficiently with large state space scenarios, however, they cannot learn from previous tasks nor improve their performance. While RL is less efficient when dealing with large state spaces, it can compensate with its learning capability. At the beginning, performance might be poor, but with time repairing becomes straightforward. Also, search and genetic algorithms require a fitness function to converge. This function is more rigid to personalize than RL rewards. While in RL is easy to adapt different rewards for individual actions or complete sequences, is not so intuitive how to provide personalization at different levels with a fitness function.

8 Conclusions and future work

In this paper, we presented an approach to repair models using Reinforcement Learning (RL) and Transfer Learning (TL) in our tool Personalized and Automatic Repair of MOdels using REinforcement Learning (PARMOREL), together with a proof of concept implementation. In this approach, experience generated by repairing models under certain customization preferences can be reused to streamline later repairs. Different parts of the experience are taken into account depending on user preferences. Each execution updates the stored experience, hence, the algorithm's learning becomes more efficient with time.

In the proof of concept implementation, we repaired broken models with different user preferences. To show how TL works under different circumstances, we simulated a set of users with preferences, such as punishing modification of model elements. The implementation showed how our approach allows us to repair models and to automatically share experience in different executions and models, achieving better performance the more experience is reused. Our results are promising and can be seen as an indicator of the potentials of this research direction, hence, we would like to continue developing PARMOREL following the next research lines.

Modeling framework. The implementation displayed in Section 5 is tied to EMF and Ecore metamodels. However, since PARMOREL is built as an Eclipse plugin we are currently working on implementing features that enable users—through implementing a series of interfaces—to define both the issues they want to repair, their own catalogue of actions and types of models.

Preferences and dynamic learning. Next, we will work further on how users define their preferences for model repair. We are developing a domain-specific language (DSL) to allow users to design their own preferences in addition to offering a predefined set of preferences (as shown in Fig. 6). Regarding rewards, we would like to apply apprenticeship learning [AN04] to infer their values from observing users during repairing processes. Furthermore, we are working on enabling the users to give feedback to the algorithm by selecting among the repair sequences provided by the algorithm. This way, the users can determine which of the solutions fit their requirements after checking the effect of the repair sequence. Moreover, we will investigate how historical changes in the models could be used to influence the final repair sequence.

Collaborative environment. Currently, PARMOREL works sequentially and concurrent sharing of experience is not supported. That is, we store experience in an XML file that can be shared via a repository. This method works as a proof of concept, however, we plan to provide a collaborative environment where experience is gathered and shared in runtime.

Quality metrics. Moreover, the only measurable quality of the repaired models is how much they fit user preferences. In future work, we want to also assure these models' quality based on metrics [DG18]. The same way we can produce personalized models by using preferences, we will be able to produce models that improve different quality metrics at request. Finally, one area we want to study is the refactoring of models using RL to make them more aligned to architectural and design patterns. Additional rewards could be related to how well the models meet the coupling and cohesion criteria.

References

[AKK+08] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. Amor–towards adaptable model versioning. In *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*, volume 8, pages 4–50, 2008.

[aml16] amlModeling. amlmodeling/amlmetamodel, Jan 2016. URL: `https://github.com/amlModeling/amlMetaModel`.

[AN04] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

[Bar] Angela Barriga. PARMOREL. Available at: `https://ict.hvl.no/project-parmorel/`.

[BBBK11] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[BDRIP19] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Quality-driven detection and resolution of metamodel smells. *IEEE Access*, 7:16364–16376, 2019.

[Bel13] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[BRH18] Angela Barriga, Adrian Rutle, and Rogardt Heldal. Automatic model repair using reinforcement learning. In *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018.*, pages 781–786, 2018. URL: `http://ceur-ws.org/Vol-2245/ammore_paper_1.pdf`.

[BRH19] 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)*, pages 175–182, 2019. [Forthcoming]. Available: https://bit.ly/2IPfwMD.

[BV10] Manuel F Bertoa and Antonio Vallecillo. Quality attributes for software metamodels. *Málaga, Spain*, 2010.

[CvBLR+17] Abel Armas Cervantes, Nick RTP van Beest, Marcello La Rosa, Marlon Dumas, and Luciano García-Bañuelos. Interactive and incremental business process model repair. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 53–74. Springer, 2017.

[DG18] Khanh-Hoang Doan and Martin Gogolla. Assessing uml model quality by utilizing metrics. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 92–100. IEEE, 2018.

[Fou] The Eclipse Foundation. Eclipse modeling project. URL: `https://www.eclipse.org/modeling/emf/`.

[KKE18] Roland Kretschmer, Djamel Eddine Khelladi, and Alexander Egyed. An automated and instant discovery of concrete repairs for model inconsistencies. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 298–299. ACM, 2018.

[KKE19] 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*, pages 113–126, 2019.

[KMW+17] Marouane Kessentini, Usman Mansoor, Manuel Wimmer, Ali Ouni, and Kalyanmoy Deb. Search-based detection of model level changes. *Empirical Software Engineering*, 22(2):670–715, 2017.

[LFGDL14] Jesús J López-Fernández, Esther Guerra, and Juan De Lara. Assessing the quality of meta-models. In *MoDeVVa@ MoDELS*, pages 3–12. Citeseer, 2014.

[mar15] markus1978. markus1978/randomemf, Dec 2015. URL: `https://github.com/markus1978/RandomEMF/`.

[MGC13] Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. Model repair and transformation with echo. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 694–697. IEEE Press, 2013.

[MJC16] Nuno Macedo, Tiago Jorge, and Alcino Cunha. A feature-based classification of model repair approaches. *IEEE Transactions on Software*

Engineering, 43(7):615–640, 2016. doi:https://doi.org/10.1109/TSE.2016.2620145.

[NRA17] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. Rule-based repair of emf models: An automated interactive approach. In *International Conference on Theory and Practice of Model Transformations*, pages 171–181. Springer, 2017.

[Occ17] Occiware. occiware/ecore, Sep 2017. URL: https://github.com/occiware/ecore/.

[OPKK18] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. Revision: a tool for history-based model repair recommendations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 105–108. ACM, 2018.

[PVDSM15] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, 14(1):461–481, 2015.

[PY10] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[RE12] Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 220–229. ACM, 2012.

[SB11] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2011.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[TOLR17] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. Change-preserving model repair. In *International Conference on Fundamental Approaches to Software Engineering*, pages 283–299. Springer, 2017.

[TS10] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. IGI Global, 2010.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

About the authors

Angela Barriga is a PhD Candidate at Western Norway University of Applied Sciences. She has experience woring with machine learning, computer vision, gerontechnology and pervasive systems. Barriga's thesis is focused on model repair, specially on repairing using reinforcement learning. She has been part of the local organization of iFM 2019 and is involved in STAF 2020-2021. She is also part of the program committee of the third international workshop on gerontechnology. You can learn more about her at https://angelabr.github.io/ or contact her at abar@hvl.no.

Adrian Rutle is a Full-time professor at Western Norway University of Applied Sciences. Adrian holds PhD in Computer Science from the University of Bergen, Norway. Rutle is professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen. Rutle's main interest is applying theoretical results from the field of model-driven software engineering to practical domains and has expertise in the development of modelling frameworks and domain-specific modelling languages. He also conducts research in the fields of modelling and simulation for robotics, eHealth, digital fabrication, smart systems and machine learning. Contact him at `adrian.rutle@hvl.no`

Rogardt Heldal is a professor of Software Engineering at the Western Norway University of Applied Sciences. Heldal holds an honours degree in Computer Science from Glasgow University, Scotland and a PhD in Computer Science from Chalmers University of Technology, Sweden. His research interests include requirements engineering, software processes, software modelling, software architecture, cyber-physical systems, machine learning, and empirical research. Many of his research projects are performed in collaboration with industry. Contact him at `rogardt.heldal@hvl.no`

PAPER B

MODEL REPAIR WITH QUALITY-BASED REINFORCEMENT LEARNING

L. Iovino, A. Barriga, A. Rutle, and R. Heldal.

In Journal of Object Technology, Volume 19, Number 2, 2020.

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Model Repair with Quality-Based Reinforcement Learning

Ludovico Iovino[b] Angela Barriga[a] Adrian Rutle[a]
Rogardt Heldal[a]

a. Western Norway University of Applied Sciences, Norway
b. Gran Sasso Science Institute - Computer Science Scientific Area, Italy

Abstract Domain modeling is a core activity in Model-Driven Engineering, and these models must be correct. A large number of artifacts may be constructed on top of these domain models, such as instance models, transformations, and editors. Similar to any other software artifact, domain models are subject to the introduction of errors during the modeling process. There are a number of existing tools that reduce the burden of manually dealing with correctness issues in models. Although various approaches have been proposed to support the quality assessment of modeling artifacts in the past decade, the quality of the automatically repaired models has not been the focus of repairing processes. In this paper, we propose the integration of an automatic evaluation of domain models based on a quality model with a framework for personalized and automatic model repair. The framework uses reinforcement learning to find the best sequence of actions for repairing a broken model.

Keywords MDE; Machine Learning; Model Repair; Quality Evaluation

1 Introduction

Models are becoming core artifacts of modern software engineering processes [WHR14]. When performing modeling activities, the chances of breaking a model increase together with the size of development teams and the number of changes in software specifications, due to lack of communication, misunderstanding, mishandled collaborative projects, etc [TOLR17]. The correctness and accuracy of these models are of the utmost importance to correctly produce the systems they represent. However, it can be a time-consuming task to make sure that models are correct and have the required quality. Therefore, several approaches to automatic model repair have been proposed in the past decades [OPKK18, NRA17, MGC13]. However, the quality of the automatically repaired models has not been the main focus of the repairing algorithms even though quality characteristics have been extensively studied in the literature [BBL76, Dro95, OPR03]. Usually, a common approach to define quality models is to first identify a small set of

Ludovico Iovino, Angela Barriga, Adrian Rutle, Rogardt Heldal. *Model Repair with Quality-Based Reinforcement Learning*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 17:1–21. doi:10.5381/jot.2020.19.2.a17

high-level quality characteristics and then decompose them into sets of subordinate characteristics. We consider customization important due to the flexibility of the concept of quality; quality characteristics may be given different meanings depending on the considered application scenarios, context, and intended purpose [BDRDR+16]. Hence in this paper, we propose the integration of an automatic model repair method with a customizable quality definition method.

In previous work, we introduced PARMOREL (**P**ersonalized and **A**utomatic **R**epair of **MO**dels using **RE**inforcement **L**earning) [BRH18, BRH19], an approach that provides personalized and automatic repair of software models using reinforcement learning (RL) [TL00]. PARMOREL finds a sequence of repairing actions according to preferences introduced by the user without considering objective measures such as quality characteristics. In this paper, we extend our approach to also consider well-known metrics to improve the overall quality of the repaired models. To achieve this, we integrate PARMOREL with a tool for quality evaluation of modelling artifacts [BDRDR+19]. This tool facilitates the evaluation of the modeling artifacts in terms of a personalized view of the quality concepts. This integration leads to the production of models that are improved based on both user preferences and quality characteristics—such as maintainability and understandability. Our approach takes automatic model repair one step further in supporting users to improve the quality of models.

Structure of the paper. This paper is organised as follows: section 2 presents a running example where we demonstrate how a domain model with errors can be repaired in different ways. In section 3, we propose some quality characteristics to be evaluated on the running example in order to demonstrate how different actions impact the repaired domain model differently. We show the proposed extended architecture of PARMOREL in section 4, evaluate the approach in section 5, and discuss factors which might affect the validity of the evaluation in section 6. In section 7, we present some relevant related works and we conclude the paper in section 8.

2 Running example

In this section, we demonstrate how a broken domain model can be repaired. We will show how different actions can lead to different resulting repaired models with different quality characteristics. As an example, we will use the model in Figure 1 that shows a domain model specified using the Eclipse Modeling Framework (EMF) [SBMP08a]. This model represents an excerpt of the "company" application domain. The root of the model is a CompanyModel containing a set of companies. A Company is defined with a name, and it can hire a set of Employees. Every employee has a name and specializes the class Person. Client is another specialization of Person. Moreover, the model can define Projects.

The chances of breaking a model increase with collaborative modeling activities, depending on the number of changes in software requirements [BRH18], and the size of the conceptual domain to be engineered. This domain model might have become invalid at any stage of the modeling activity [1]. The model in Fig. 1 presents a number of problems as seen from the highlighted parts (in red). The invalid model is unsuitable

[1] This domain model has been taken from a dataset of academic examples used during the MDE Course at the GSSI, Italy. This model has been created by junior MDE experts, during the lab-sessions of the course.

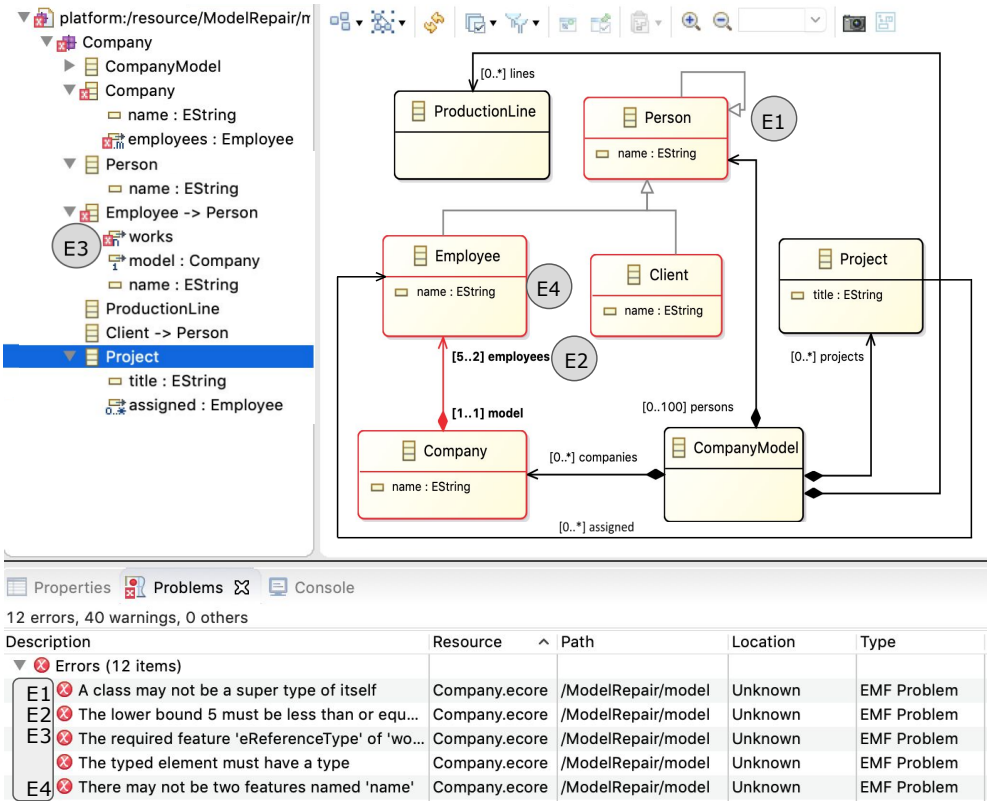


Figure 1 – A snapshot of an invalid domain model

for modeling activities that require model validation [MLLD10]. We summarize these problems as follows:

E1 Person cannot inherit from itself

E2 Reference employees of the class Company has lowerbound greater than upperbound

E3 Reference works of the class Employee is untyped (see treeview based representation)

E4 There cannot be two features name in the same class (also including inherited attributes).

These errors can be repaired in multiple ways [DREI+16] leading to different results but with the same intent—restore the validity. For each error, a possible set of resolution actions can be undertaken. To avoid increasing the search space and the complexity of the example, we consider only two possible actions per error (these actions were chosen manually with an illustrative purpose). We group these actions into two sets: A1 and A2. Furthermore, the domain models in Fig. 2 and Fig. 3, respectively, show the effects of repairing the errors applying A1 and A2.

Starting from error E1 where a class cannot have itself as a supertype, the resolution actions we propose are:

A1 Removal of the supertype relationship

A2 Adding a new class to satisfy the supertyping

In this specific case the Person class having a supertype relationship to itself can be resolved with A1 where we remove the relationship or with A2 where we introduce a new class where its name is $super\{class - name\}$, i.e. SuperPerson.

Error E2, where the lower bound of the employees reference is greater than the upper bound can be solved with one of the following actions. Concretely, the two actions can result in modifying employees cardinality [5..2] to [2..5] or [1..2] (by decreasing lowerBound until it is smaller than the upperBound value: 2).

A1 Invert lowerBound with upperBound of the reference

A2 Decrease lowerBound or increase upperBound of the reference until the model results valid

For E3 where the works reference is untyped, the possible resolution actions are numerous if we decide to pick any class in the model to type the reference. To reduce the search space, we apply a heuristic that only allows to type references by classes having less than 2 ongoing references. In our example, only classes Project, ProductionLine and Client comply to this heuristic. To maintain the uniformity of presenting 2 actions per error, we concentrate on the first two classes; typing works with Client do not produce any significative changes w.r.t. the actions proposed below:

A1 Type it with Project (merging works with the existing reference assigned into a bi-directional reference)

A2 Type it with ProductionLine

The last error to be fixed is E4 where the name attribute cannot be repeated since the superclass of Employee and Client already has declared it.

In this case the resolution can be listed as follows:

A1 Remove the attribute from the supertype

A2 Remove the attribute from the subtypes

These two actions applied to the domain model resulted in having the attribute name in the Person class or in both of its subclasses Client and Employee. Also, in this case, we could propose other alternative actions, but what is important is the concept that we will detail in the next section.

3 Quality evaluation

In [BDRDR$^+$19, BDRDR$^+$16, LFGDL14] different works propose quality models specifically conceived to measure the quality of models and other modeling artifacts. In these works, characteristics like maintainability, portability, and usability are introduced together with sub-characteristics like analyzability, adaptability, and understandability for each main characteristics. Multiple quality characteristics may be considered in order to evaluate qualitative aspects of the domain engineering phase, formalized as a domain model. This aspect is particularly relevant in the activity of model repair since the actions undertaken to repair the errors could produce valid models, but with

low-quality characteristics. The alternative is to manually repair models with the disadvantage of being very time consuming. For this reason, we dedicate this section to demonstrate how the two actions selected for each error (exposed in section 2) may produce valid domain models but with different quality characteristics.

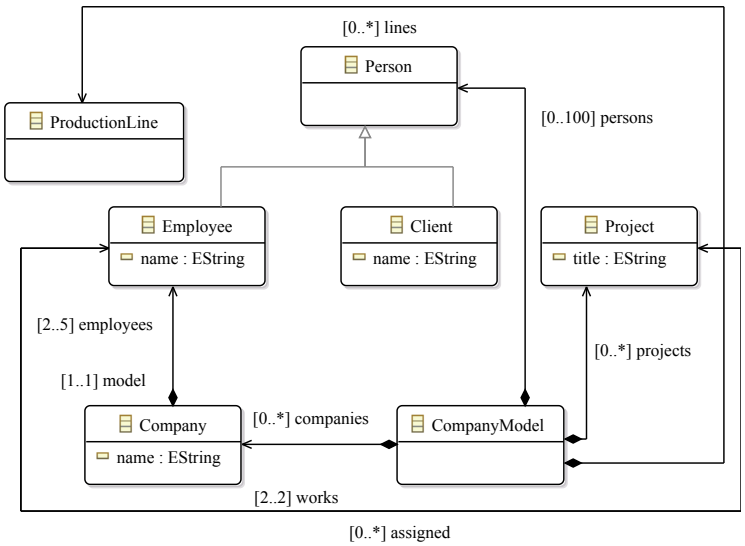


Figure 2 – Repaired model with actions A1

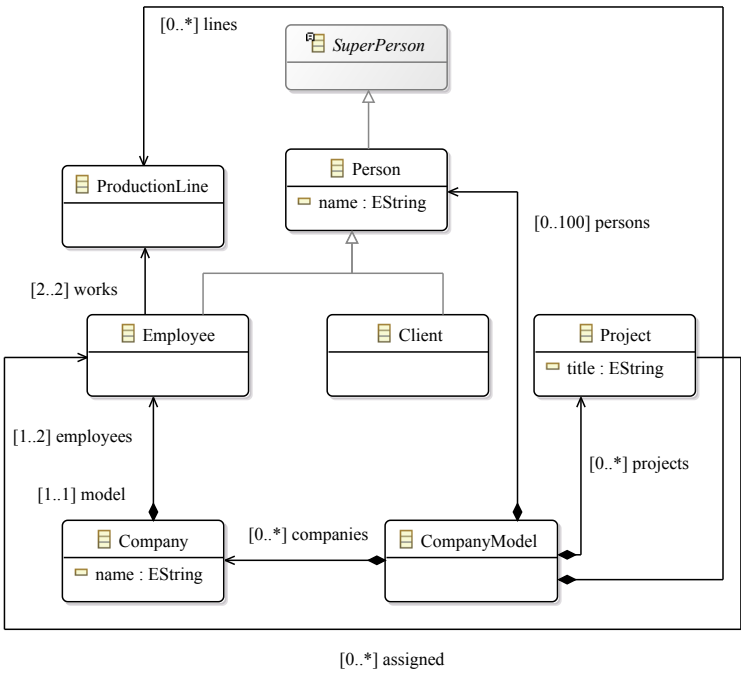


Figure 3 – Repaired model with actions A2

3.1 Quality characteristics

In this section, we consider the following quality characteristics [GP01]: maintainability, understandability, complexity, and reusability. For measuring the quality characteristics of the two produced domain models in Fig. 2 and Fig. 3, we have implemented a quality assessment tool inspired by [BDRDR+16]. This tool will be presented in section 4 in the overall integrated approach.

The *maintainability* quality characteristic considered in this paper has been defined according to the definition given in [GP01] and that is based on some of the metrics shown in Table 1 as follows:

$$Maintainability = \left(\frac{NC + NA + NR + DIT_{Max} + Fanout_{Max}}{5} \right) \quad (1)$$

According to the considered definition of *maintainability* the lower values the better.

The definitions of the *Understandability* and *Complexity* quality characteristics are adopted from [SC06]. In particular, understandability can be defined as follows:

$$Understandability = \left(\frac{\sum_{k=1}^{NC} PRED + 1}{NC} \right) \quad (2)$$

where PRED regards the predecessors of each class, since, in order to understand a class, we have to understand all of the ancestor classes that affect the class as well as the class itself. According to such a definition, the lower values for the understandability quality characteristic the better.

Complexity can be defined in terms of the number of static relationships between the classes (i.e., number of references). The complexity of the association and aggregation relationships is counted as the number of direct connections, whereas the generalization relationship is counted as the number of all the ancestor and descendant classes. Thus, the complexity quality characteristic can be defined as follows:

$$Complexity = (NR - NUR + NOPR + UND + (NR - NCR)) \quad (3)$$

Metric	Acronym
Number of Class	NC
Number of TotalReference	NR
Number of Opposite Reference	NOPR
Number of TotalReference containment	NCR
Number of TotalAttribute	NA
Number of Unidirectional reference	NUR
Max generalization hierarchical level	DITmax
Max Reference Sibling (max fan Out)	FANOUTmax
Number of TotalFeatures	NTF
Sum of inherited structural features	INHF
Attribute inheritance factor	AIF
Number of predecessor in hierarchy	PRED
Within an relation chain is the longest path from the class to others	HAGG
Difference between the upper bound and lower bound in a reference	REFint
Max or min upper bound of a set of references	UPBmax|UPBmin

Table 1 – Excerpt of the metrics considered in the evaluation

Quality characteristics	A1	A2
Maintainability	4.20	4.60
Understandability	1.28	1.62
Complexity	12.28	8.6
Reusability	0.00	0.15
Relaxation Index	4.57	4.56

Table 2 – Quality characteristics after evaluation of the running example (table)

where NUR is the number of unidirectional references calculated as the difference between bidirectional and total reference number, and UND is the understandability value calculated as defined in Def. 2. According to the given definition, the lower values for the complexity characteristic the better.

The *reusability* of a given model can be calculated in different ways. One of these is to use the attribute inheritance factor *AIF* as proposed in [Are14] where it is stated that a higher value indicates a higher level of reuse. As presented in [AJS07], AIF can be defined as follows:

$$Reusability = AIF = \left(\frac{INHF}{NTF}\right) \tag{4}$$

where *INHF* is the sum of the inherited features in all classes, and *NTF* is the total number of available features.

Moreover, we decided to define a new quality characteristic inspired by the concept of metamodel relaxation [AA17], called *relaxation index*.

$$RelaxationIndex = \left(\frac{\sum_{k=1}^{NR} REFint - UPBmin}{UPBmax - UPBmin}\right) \tag{5}$$

Based on the concept of relaxation we can define how much a relation is strict with respect to its cardinality constraints. For instance, [0..*] on a reference is more relaxed compared to [i..i] (for $i \in INT$). This is because in the first case the modeler has more freedom to define the number of instances, that can be optional or even infinite; in the second case he / she needs to define exactly `i` instances to fulfill the constraints. For this reason, if we want to give more elasticity to the modeler, we can use this index to understand which model is less restrictive.

3.2 Evaluating the running example

When we calculate these quality characteristics on the running example, the result can be summarized as follows (see Table 2).

The application of the two actions can have different impact on the quality characteristics of the domain models. In fact, it seems that A1 resulted better in maintainability, understandability and relaxation index, while A2 resulted better in the rest. Maintainability is calculated over the number of model elements, the hierarchical definitions and the siblings of every element and the balance of these elements has made the result similar for the two actions, even if A1 resulted better.

In the same way, understandability resulted better for A1, since also this quality characteristic is based on the predecessors of the classes. Complexity resulted better in A2, being partially linked to the understandability, but also to the unidirectional and bidirectional references. In fact A1 introduces a reference with type Project (works),

this class already has a unidirectional reference to Employee (assigned), matching then a new eOpposite constraint, i.e. bidirectional, making the model more complex.

Reusability indicates that A2 produces a model with a better level of reuse. This is due to the fact that A2 concerning the name attributes decides to move it up to the hierarchy, instead of maintaining the ones in the subtypes. This increases the number of inherited features and hence the level of reuse. Finally, considering the relaxation index the A1 set of actions results in generating a domain model slightly more relaxed with respect to the A2 set. In fact, the references are all the same, except the employees relation where in case of A1 is set to [2..5], and in A2 to [1..2], affecting the relaxation index.

4 Customizing quality characteristics with RL

Up to now, the results of our quality evaluation are based on definitions taken from existing literature (see Section 3.1). In this section, we demonstrate how the quality definition may be specified and customized by the modeler based on her own point of view on model quality. Our approach takes the quality preferences from the modeler as input and uses Reinforcement Learning (RL) to find customized repairing sequences of actions that improve the selected characteristics.

PARMOREL uses RL algorithms (currently, Q-learning [TL00]) to find which is the best possible repairing action for each error in the model. RL consists of algorithms able to learn by themselves how to interact in an environment without existing pre-labelled data, only needing a set of available actions and rewards for each of these actions. We rely on an external modeling framework (i.e. the Eclipse Modeling Framework (EMF) [SBMP08b]) to retrieve issues in the models (e.g. attribute without a type, duplicated class). The modeling framework is also responsible for applying the actions selected by PARMOREL and creating the repaired models. The learning algorithm in PARMOREL allows to provide repairing from zero, without knowing any details of the model to be repaired. By using and tuning RL rewards, these algorithms can learn which are the best actions to repair a given error. We can adapt these rewards to align with any preference introduced by the user, including quality characteristics; i.e., if a user prefers to improve maintainability in the model, we assign positive rewards to actions that satisfy this quality metric (note that it is mandatory for users to introduce, at least, one metric preference).

Figure 4 shows the extended workflow of PARMOREL. Before finding a repairing sequence for a given model, PARMOREL is executed for a number of episodes. Each episode equals one iteration attempting to repair the model, as reported with dotted lines in Figure 4. For each of these episodes, a possible repairing sequence is found, and applying it, a provisional repaired model is created.

By introducing quality evaluation to PARMOREL, we can measure the quality for each of the provisional repaired models. PARMOREL translates these results into rewards, so that it can identify how good each repairing action is improving or preserving the considered quality of the model. Likewise, after each episode, only actions providing higher quality would be selected. This is done by extending the architecture to include the *Quality Evaluation Module*, that we will detail in the remaining of this section. After performing enough repairing iterations, PARMOREL will select the repair sequence with higher rewards and saves the final repaired model.

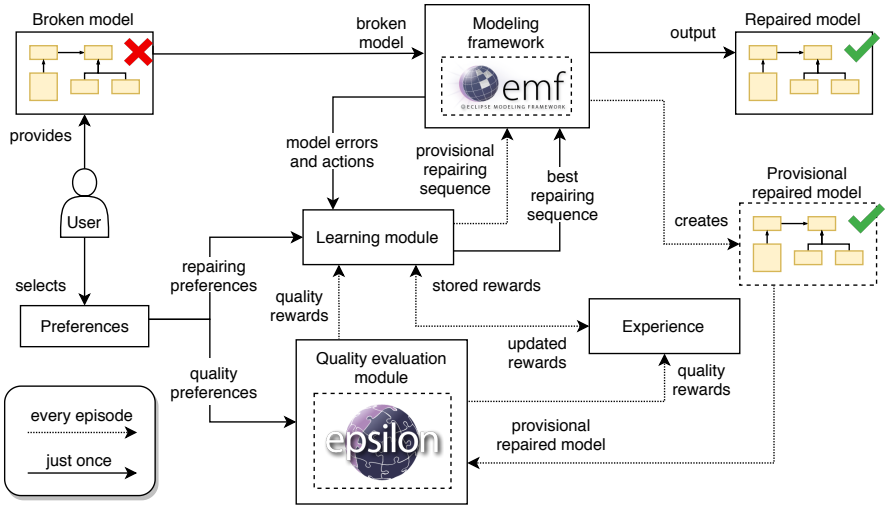


Figure 4 – PARMOREL's workflow for assuring quality in repaired models

4.1 Specification of quality aspects

The Quality model plays a key role in the proposed approach since it enables the specification of quality measures according to the domain's or the modeler's requirements. Inspired by the quality model proposed in [BDRDR+16], we have designed a model for the specification of the quality of multiple artifacts (see Fig. 5). Each of these artifacts will be assigned a set of QualityCharacteristicss in which the modeler can specify, among others, the calculation function (functionaName) and the priority with respect to other quality characteristics. Moreover, whether a quality characteristic should be maximized or minimized, is specified in the attribute solution.

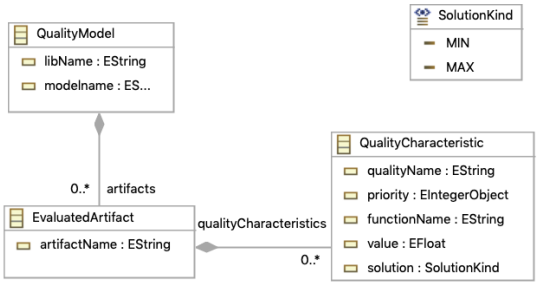


Figure 5 – Quality characteristics model

Furthermore, the attribute value—which is empty at the beginning—will be actualized with the resulting value of the evaluation by the engine which executes the quality calculation function; this function is presented as a workflow diagram in Fig. 6. Indeed, the modeler specifies the initial setting of the quality characteristics according to her preferences while the engine applies the calculation function on the given artifacts to determine their quality.

Figure 6 reports a simplified representation of the quality evaluation process. For each provisionally repaired model—i.e., after applying the repairing algorithm once—, the evaluation engine will be invoked on two inputs: the quality preferences

which is an instance of the model in Fig. 5 and the provisional model which is subject for the evaluation. The evaluation engine will actualize the QualityCharacteristics's attribute value in the quality model. Then the results become available for inspecting the values of the calculated quality characteristics. These results will be used by PARMOREL to optimize the required qualities in subsequent episodes.

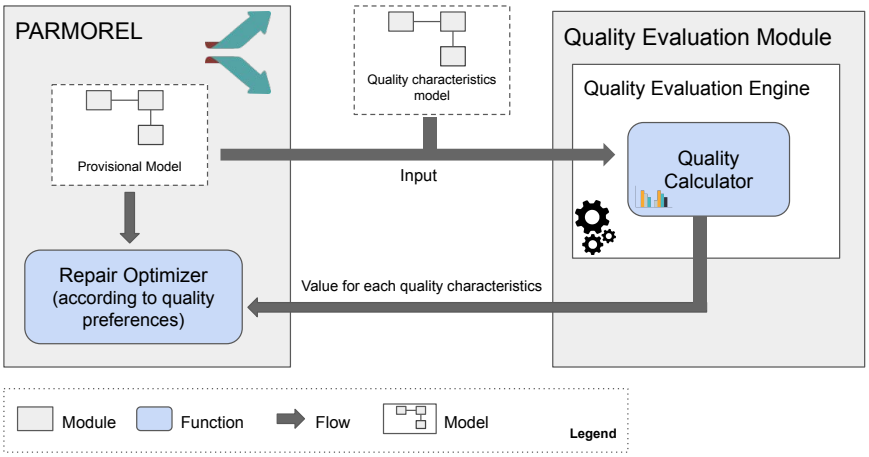


Figure 6 – Excerpt workflow of quality characteristic evaluation

The evaluation engine has been implemented with EOL [KPP06], an imperative programming language for creating, querying and modifying EMF models. EOL offers model management operations with a dedicated language built on top of EMF. This makes easier the definition of evaluation operations with respect to Java implementations using EMF API directly [BDRDR+16]. A declared library is used to evaluate the domain models given as input. An excerpt of this specification is reported in Algorithm 1, which is an abstraction of the EOL library.

Algorithm 1 Quality evaluation, EOL main file excerpt

```
1: IMPORT: qualityModel as QM
2: INPUT: provisionalModel as MM
3: QM.evaluatedArtifact ← MM
4: maintainability ← (n_classes(MM) + n_attrs(MM) +
          n_refs(MM) + dit_max(MM) + hagg_max(MM))/5
5: ...
6: //if maintainability is declared in the quality model
7: if QM.maintainability != ∅ then
8:     QM.maintainability ← maintainability
9:     QM.evaluatedArtifact.addCharacteristics(maintainability)
```

First, the evaluation begins by setting the evaluated artifact (line 3) with the domain model passed as provisional model in Fig. 6. Further, all the quality characteristics declared in the quality model will be evaluated. For instance, line 7 evaluates if maintainability is declared in the quality model given as input to the evaluation (line 1). A representative quality model is depicted in Fig. 7, where the modeler has declared the five quality characteristics to be evaluated (anticipated in section 2), and set the functionName to the name of the function used in the EOL script to invoke the calculator. Algorithm 1 reports only one of the quality characteristics available in that

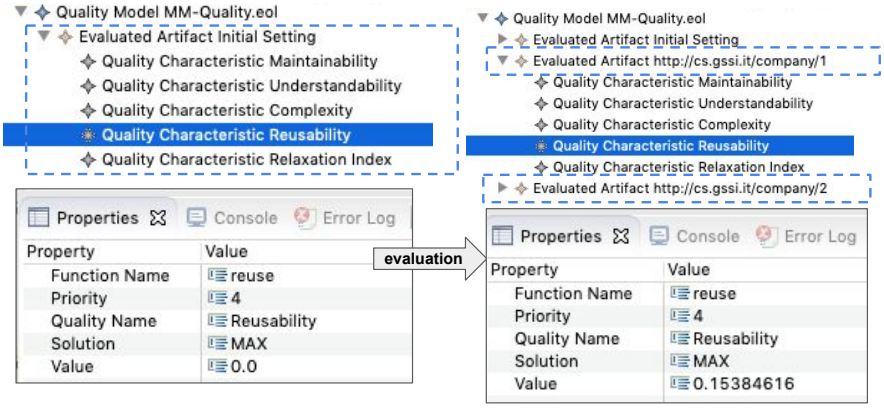


Figure 7 – Quality model initial setting and result

library (maintainability), but the modeler can evaluate others by simply declaring them as model elements, as in Fig. 7 (left).

The evaluation of the given artifact will refine the artefact's existing quality model. In the initial setting, the modeler selects the quality characteristics to be evaluated with a specific user-defined priority. As seen in Fig 7 (right), a new evaluation will be available for each domain model under evaluation, and every quality characteristic has its related value actualized. Here, we highlight the evaluated domain model given as input to the process, identifiable with the unique URI http://cs.gssi.it/company/1 of the domain model. In the property view the reusability evaluation of the domain model obtained with the application of A1 shows the reuse value 0.153.

4.2 Using quality evaluation in preference specification

Using the initial setting of the quality model, as for instance the one depicted in Fig. 7, the modeler can instruct the reward algorithm. To achieve this, the modeler can select which quality characteristics he wants to measure in the model and in which priority.

First, PARMOREL calculates a reward for each of the quality characteristics selected (see lines 3 - 6 in Alg. 2). This is done by subtracting the value of the quality characteristic of the original model from the provisional repaired version. The order of the subtraction is altered depending on whether the quality characteristic should be maximized or minimized. Then, the algorithm multiplies each reward value by the corresponding quality characteristic priority value. As a consequence, we obtain stronger rewards for the characteristics which the modeler considers of higher relevance. The rewards values are normalized in order to avoid big numerical differences when one of the quality characteristics varies more than the others.

Finally, PARMOREL adds all the rewards and stores the obtained value for each action in the selected sequence in the *Experience module*. Following this procedure, after each episode PARMOREL will be able to produce repaired models of higher quality since the algorithm will progressively apply actions with higher rewards. Thanks to the random component of PARMOREL's Q-learning, the algorithm will also be able to apply new actions that otherwise would not be selected due to the other actions having already higher rewards. This random component assures the discovery of different repairing sequences that might lead to higher quality models.

Algorithm 2 Rewards calculation in PARMOREL

```
1: INPUT: from Modeler (qualitycharacteristics, originalModel)
2: INPUT: from PARMOREL (repairedModel, sequenceActions)
3: for each qa in qualitycharacteristics do
4:     reward ← getQuality(qa, originalModel) - getQuality(qa, repairedModel)
5:     reward ← reward * qa.priority
6:     rewardsList ← reward
7: normalize(rewardsList)
8: experienceModule(repairedModel, rewardsList, sequenceActions)
```

Figure 8 displays the results of repairing the model from Fig. 1 by using three of the quality characteristics introduced in Section 3: complexity, reusability and understandability. Working with these quality characteristics is especially interesting, since improving complexity and reusability involves reducing the number of elements, which might lead into getting a worse value for understandability. Here we show how RL can make a compromise in order to satisfy all characteristics as much as possible.

The four initial syntactical errors (E1-E4) are repaired with the available actions presented in Section 2, where we showed an example of how would the repair be when applying all A1 actions (see Fig. 2) or A2 actions (see Fig. 3). In this new example, PARMOREL picks the actions that achieve better results for all characteristics: E1 is repaired with its correspondent A1 and E2-E4 are repaired with their A2s. The reasoning behind choosing these actions rely on minimizing the number of elements in the model with a special focus on hierarchies, which boosts the understandability.

PARMOREL finds different solutions depending on the considered quality. For example, Fig. 9 displays the results of repairing the model from Fig. 1 when prioritizing another three quality characteristics from Section 3: maintainability, reusability and relaxation index. This time, PARMOREL picks the following actions: E1-E3 are repaired using their A1s and E4 is repaired with A2. With these actions the produced model has less elements, which improves the maintainability and reusability and the employees reference has more relaxed bounds.

In this section, we introduced a scenario which highlights how PARMOREL can be instructed to consider the user preferences, specified by a quality model.

5 Evaluation of PARMOREL

To evaluate and test the scalability of our approach, we use PARMOREL to repair 107 domain models[2] and consider two research questions:

RQ1 How the size of the model affects the execution-time of the repair?

RQ2 How the number of errors in the model affects the execution-time of the repair?

To answer these questions, we conducted an experiment using syntactically corrupted models and two metrics: number of errors and number of elements (size). We split the dataset of models with an 80-20% distribution, repairing 20% of the models twice, with and without having first repaired the 80%. With this experiment, we analyze the impact of number of errors and size on the repairing time of the 80% and the influence these metrics have when reusing learning and streamlining the repair on the 20%.

[2]This dataset is available on this Git repository: https://github.com/MagMar94/ParmorelRunnable

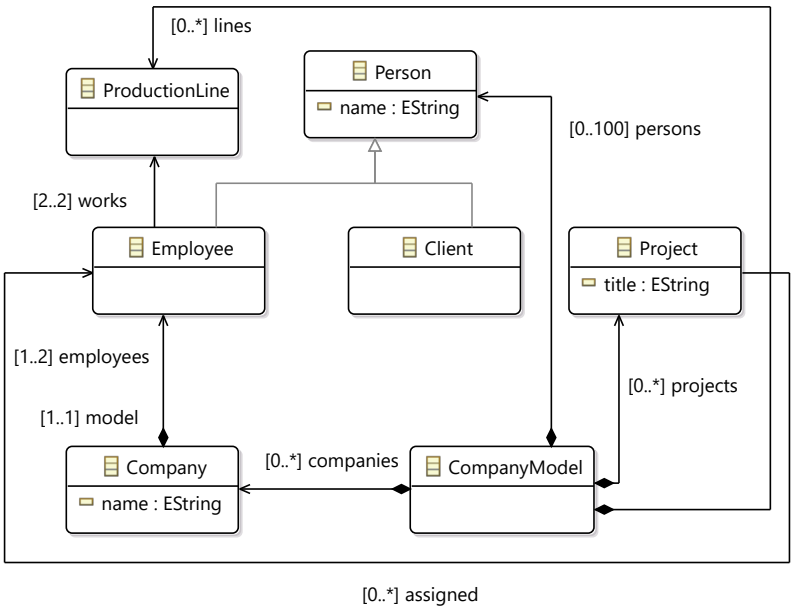


Figure 8 – Model from Fig. 1 prioritizing complexity, reusability and understandability

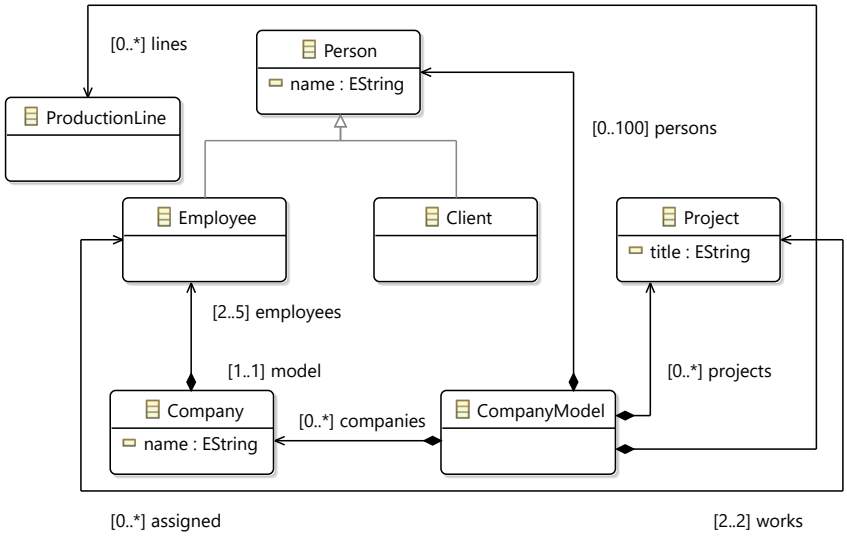


Figure 9 – Model from Fig. 1 prioritizing maintainability, reusability and relaxation index

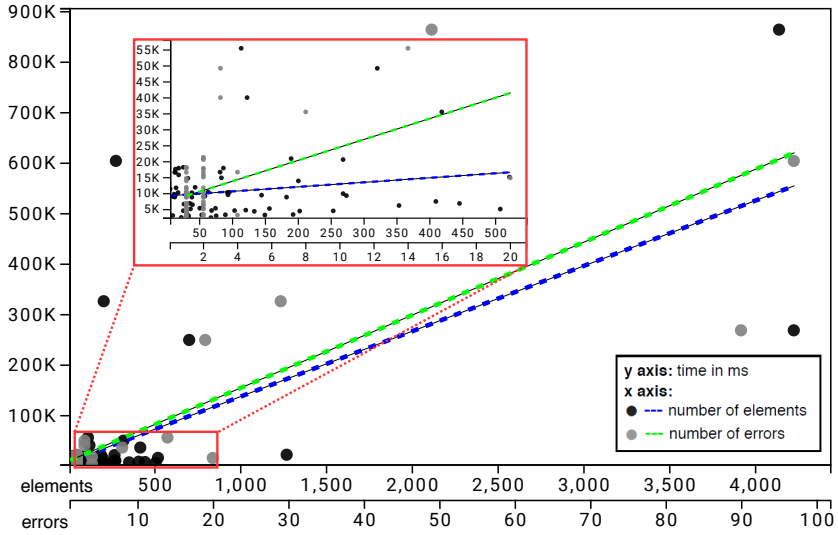


Figure 10 – Relation between repair time in ms per model size and number of errors

To retrieve these models we rely on the dataset used in [NDRDR+19] and filtered in order to get only corrupted Ecore models. All errors present in these models are syntactic errors that violate certain constraints of the Ecore metamodeling language [SBMP08b] (e.g., the opposite of the opposite of a reference must be the reference itself, classifiers must have different names, etc). Each subject model contains between 1 and 118 errors, counting a total of 12 different types of errors throughout the models. Regarding the number of elements, each model has between 12 and 4227, counting the number of classes, attributes, references, and operations.

In the following, we present the results of the experiment. First, we configure PARMOREL to run 25 episodes to repair each model in 80% of the dataset. There is no established policy of how many episodes are best for a given problem [TL00], so according to our experimentation, 25 are enough to find at least one repairing sequence of actions for every model. The execution of all the episodes takes between 2.1s (for a model M1 with 70 elements and 1 error) and 14.38 mins (for a model M2 with 4141 elements and 49 errors—the second biggest model in the dataset); this is shown in Fig. 10. We include a zoomed area in Fig. 10 to display the details of models with less than 550 elements and 20 errors since these constitute the majority of the 80% dataset. The number of errors influences the repair time slightly more than the size of the models since PARMOREL needs to go throughout each model structure to find and repair each error, so the more errors are in the model the more calculations are required. The total execution time for the 80% dataset is 51.35m.

We can conclude that both the size of the models and the number of errors affect the repairing time logarithmically (see Fig. 10), although the influence of the latter is stronger. The influence of these factors is confirmed by calculating the correlation coefficient [ASG06] for each pair of values: we get a coefficient of 0.68 for size/time and 0.80 for number of errors/time.

Next, we focus on testing the impact of the number of errors and model size when PARMOREL reuses learning from previous repairs. Additionally, we measure how much the repair is streamlined. This process can be conceived as the usual training phase in other ML algorithms. When reusing learning, the process needs fewer episodes

to converge since the tool has acquired knowledge from previous repairs. Hence, we configure PARMOREL to reduce the number of episodes by 50%, making a total of 12 episodes. First, we proceed to repair the remaining 20% of the dataset directly after repairing the previous 80%. Then, we repair again the 20% after resetting the Qtable, this is, deleting the learning obtained from the 80% repair. By comparing the results from these two rounds, we can conclude that PARMOREL streamlines the repairing time of the new models between 3% and 84% when it has learned from repairing other models. Faster repairing happens in models with bigger size, since the bigger the models, the more learning can be reused from previous repairs. On average, there is an improvement of 66.65% on the repairing time of the 20% set (without previous learning: 18.17m, reusing learning: 6.06m). With this experiment, we can conclude that repairing time depends more on the number of errors but performance improvement on the size of the models.

We could see from our testing that different quality characteristics do not alter the timing results; the algorithm gets different rewards and therefore the produced repaired models will be different, however, the time required to repair them will remain the same. For this reason, in this section, we only presented the repair which aims to boost the maintainability quality characteristic.

The results of this evaluation indicate that PARMOREL is scalable and that it can handle real-world corrupted models. Furthermore, the approach works with models with different amounts and types of errors, finding a repairing solution for all of them.

6 Threats to Validity

In this section, we comment the threats to validity of our research, following the guidelines from [WRH+12].

Internal threats. Quality evaluation may be considered as an internal threat to validity since the quality model is user-defined. The definition of quality aspects tends to be based on the user's experience. Moreover, mistakes in these definitions could lead to faulty results. This can be partially mitigated by including quality experts in the definition process, or, as in our experiments, by relying on definitions based on formulae in the literature. Also, the result of combining different quality characteristics could lead to results not aligned with the goals of a user. Again, this can be avoided by including experts who can guide which characteristics should (or not) be combined.

External threats. A potential external threat to the validity of our evaluation is the dataset used for the experiments. We have selected corrupted models resulting in a dataset of 107 models, which may be considered small, however, 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.

Also, throughout the paper we have picked five characteristics (maintainability, understandability, reusability, complexity, and relaxation index) as a proof of concept to show the potentials of PARMOREL and used one of them, maintainability, for the evaluation. We consider this sample set representative enough for our experiments but the approach is not limited to this set since it supports any characteristics defined using the Epsilon Language. Although the implementation displayed in Section 5 is tied to EMF and Ecore models, PARMOREL is built as an Eclipse plugin, so it is possible to use other modelling frameworks—through implementing a series of interfaces—and users can define both the issues they want to repair and their own catalogue of actions and types of models.

7 Related work

Over the years, various approaches have been proposed to support the quality measurement of modeling artifacts using quality models. The authors in [BDRDR+16, BDRDR+19], propose quality models [Are14] to measure the quality of modeling artifacts. A number of tools have been developed to support quality evaluation in UML [AT16] or EMF [AST10]. Others focus on quality evaluation of valid models, with the intent of applying refactorings in order to improve the quality [BDRDR+19, AT13]. Although our quality evaluation builds on top of the works mentioned here, our work is different since we focus on the combination of automatic model repair and improvement of model quality.

The main feature that distinguish our approach from other model repair approaches is the capability to learn from each repaired model in order to streamline the performance. We could not find in the literature any research applying RL to model repair. The most similar work to ours we could find is [PVDSM15], where Puissant et al. present Badger, a tool based on an artificial intelligence technique called automated planning. Badger generates plans that lead from an initial state to a defined goal, each plan being a possible way to repair one error. We prefer to generate sequences to repair the whole model, since some repair actions can modify the model drastically, and we consider it counter-intuitive to decide which action to apply without knowing its overall consequences, additionally, RL performs better after each execution.

Nassar et al. [NRA17] propose a rule-based prototype where EMF models are automatically completed, with user intervention in the process. Our approach allows for more autonomy since quality preferences are only introduced at the beginning of the repair process—not during the process.

Taentzer et al. [TOLR17] present a prototype based on graph transformation theory for change-preserving model repair. The authors check operations performed on a model to identify which ones caused inconsistencies and apply the correspondent consistency-preserving operations, maintaining already performed changes on the model. Their preservation approach is interesting, however it only works assuming that the latest change of the model is the most significant.

It is worth mentioning search-based and genetic algorithm-based approaches since, although they have not been applied yet to model repair, they are possible competitors to RL. These techniques have showed promising results dealing with model transformations and evolution scenarios, for example in [KMW+17] authors use a search-based algorithm for model change detection. These algorithms deal efficiently with large state spaces, however they cannot learn from previous tasks nor improve their performance. While RL is, at the beginning, less efficient in large state spaces, it can compensate with its learning capability. At the beginning performance might be poor, but with time repairing becomes straightforward. Also, search and genetic algorithms require a fitness function to converge. This function is more rigid to personalize than RL rewards. While in RL it is easy to adapt different rewards to quality criteria, is not so intuitive how to provide personalization with a fitness function.

Lastly, another search-based approach is presented by Moghadam et al. in [MÓC11]. In this work, authors present Code-Imp, a tool for refactoring Java programs based on quality metrics that achieves promising results at code-level by using hill-climbing algorithms [SG06]. These algorithms are interesting to find a local optimum solution but they do not assure to find the best possible solution in the search space (the global optimum). By using RL we assure to find the global optimum: the sequence of repairing actions that maximize the selected quality characteristics the most.

8 Conclusions and future work

Analogous to any other software artifact, domain models are living entities and are exposed to errors. It is crucial to keep these models free of errors and assure their quality. To deal with these issues, we have developed PARMOREL, a framework for personalized and automatic model repair, which uses reinforcement learning to find the best sequence of actions for repairing a broken model according to preferences chosen by the user. In this paper, we extended PARMOREL with a quality assurance mechanism based on a quality model. We presented a motivating example demonstrating the usefulness of the approach in modeling and how this can lead to better repaired solutions. Furthermore, we evaluated the approach on a set of real-world models, achieving promising results.

In the near future, we plan to test the framework with a more extended dataset of domain models and errors, with the help of modelers that may attest if the repaired sequence really offers better quality of the repaired domain model. In particular we plan to test the presented approach with a bigger dataset of domain models coming from GitHub repositories, in order to validate the approach with real examples. Additionally, we plan to create a benchmark with the mentioned dataset, with which we will compare PARMOREL to other existent model repair approaches.

So far, we have not conducted a comparative study changing the ML algorithm of PARMOREL. This is due to the difficulty of applying ML to the model repair problem. Most well-known ML algorithms depend on large amounts of labelled data to learn how to repair a problem [MRT18]. This is a challenge in the modeling domain since available model repositories (like [KC13, BDRDR+14]) only offer unlabelled data limited in terms of size and diversity. This situation reduces the options to those algorithms within the RL domain. Alternatives to Q-learning are either too simple in terms of structure for our problem (e.g., armed bandits, Monte Carlo) or would add extra complexity that is not necessary (e.g., off-policy approaches, Deep RL methods). For further details on these examples we refer the reader to [TL00]. What we plan to do in the future is a comparative study with the automatic repairing tools presented in Section 7, paying especial attention to search-based, rule-based and automated planning approaches. Lastly, in this direction, we will work on optimizing the repair with a focus on achieving state-of-the-art time.

References

[AA17] Sanaa Alwidian and Daniel Amyot. Relaxing metamodels for model family support. In *11th Workshop on Models and Evolution (ME 2017)*, volume 2019, pages 60–64. CEUR-WS, 2017.

[AJS07] J Al-Ja'Afer and K Sabri. Metrics for object oriented design (mood) to assess java programs. Technical report, King Abdullah II school for information technology, University of Jordan, Jordan, 2007.

[Are14] Thorsten Arendt. *Quality Assurance of Software Models - A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project*. PhD thesis, University of Marburg, 2014. URL: `http://archiv.ub.uni-marburg.de/diss/z2014/0357`.

[ASG06] Agustin Garcia Asuero, Ana Sayago, and AG Gonzalez. The correlation coefficient: An overview. *Critical reviews in analytical chemistry*, 36(1):41–59, 2006.

[AST10] Thorsten Arendt, Pawel Stepien, and Gabriele Taentzer. Emf metrics: Specification and calculation of model metrics within the eclipse modeling framework. In *of the BENEVOL workshop*, 2010.

[AT13] Thorsten Arendt and Gabriele Taentzer. A tool environment for quality assurance based on the eclipse modeling framework. *Autom. Softw. Eng.*, 20(2):141–184, 2013. URL: `https://doi.org/10.1007/s10515-012-0114-7`, `doi:10.1007/s10515-012-0114-7`.

[AT16] Tamás Ambrus and Melinda Tóth. Tool to measure and refactor complex uml models. In *Fifth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications SQAMIA 2016*, 2016.

[BBL76] Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[BDRDR+14] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. Mdeforge: an extensible web-based modeling platform. In *CloudMDE@ MoDELS*, pages 66–75, 2014.

[BDRDR+16] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. A customizable approach for the automated quality assessment of modelling artifacts. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 88–93. IEEE, 2016.

[BDRDR+19] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. A tool-supported approach for assessing the quality of modeling artifacts. *Journal of Computer Languages*, 51:173–192, 2019.

[BRH18] Angela Barriga, Adrian Rutle, and Rogardt Heldal. Automatic model repair using reinforcement learning. In *International Workshop on Analytics and Mining of Model Repositories (AMMoRe), co-located with MODELS*, pages 781–786, 2018.

[BRH19] Angela Barriga, Adrian Rutle, and Rogardt Heldal. Personalized and automatic model repairing using reinforcement learning. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, pages 175–181, 2019. URL: `https://doi.org/10.1109/MODELS-C.2019.00030`, `doi:10.1109/MODELS-C.2019.00030`.

[DREI+16] Davide Di Ruscio, Juergen Etzlstorfer, Ludovico Iovino, Alfonso Pierantonio, and Wieland Schwinger. Supporting variability exploration and resolution during model migration. In *Modelling Foundations and Applications, LNCS, volume 9764*, pages 231–246, Cham, 2016. Springer International Publishing.

[Dro95] R. Geoff Dromey. A model for software product quality. *IEEE Transactions on software engineering*, 21(2):146–162, 1995.

[GP01] 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.

[KC13] Bilal Karasneh and Michel RV Chaudron. Online img2uml repository: An online repository for UML. In *EESSMOD@ MoDELS*, pages 61–66, 2013.

[KMW+17] Marouane Kessentini, Usman Mansoor, Manuel Wimmer, Ali Ouni, and Kalyanmoy Deb. Search-based detection of model level changes. *Empirical Software Engineering*, 22(2):670–715, 2017.

[KPP06] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006.

[LFGDL14] Jesús J López-Fernández, Esther Guerra, and Juan De Lara. Assessing the quality of meta-models. In *MoDeVVa@ MoDELS*, pages 3–12. Citeseer, 2014.

[MGC13] Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. Model repair and transformation with echo. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 694–697. IEEE Press, 2013.

[MLLD10] Ludovic Menet, Myiam Lamolle, and Chan Le Dc. Incremental validation of models in a mde approach applied to the modeling of complex data structures, lncs, volume 6428. In *OTM Confederated International Conferences - On the Move to Meaningful Internet Systems: OTM 2010 Workshops*, pages 120–129, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[MÓC11] Iman Hemati Moghadam and Mel Ó Cinnéide. Code-imp: a tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 41–44, 2011.

[MRT18] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[NDRDR+19] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. Automated classification of metamodel repositories: A machine learning approach. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 272–282. IEEE, 2019.

[NRA17] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. Rule-based repair of EMF models: An automated interactive approach. In *International Conference on Theory and Practice of Model Transformations*, pages 171–181. Springer, 2017.

[OPKK18] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. Revision: a tool for history-based model repair recommenda-

tions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 105–108. ACM, 2018.

[OPR03] Maryoly Ortega, María Pérez, and Teresita Rojas. Construction of a systemic quality model for evaluating a software product. *Software Quality Journal*, 11(3):219–242, 2003.

[PVDSM15] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, 14(1):461–481, 2015.

[SBMP08a] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SBMP08b] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SC06] Frederick T. Sheldon and Hong Chung. Measuring the complexity of class diagrams in reverse engineering. *Journal of Software Maintenance*, 18(5):333–350, 2006. URL: `https://doi.org/10.1002/smr.336`, `doi:10.1002/smr.336`.

[SG06] Bart Selman and Carla P Gomes. Hill-climbing search. *Encyclopedia of cognitive science*, 2006.

[TL00] Sebastian Thrun and Michael L Littman. Reinforcement learning: an introduction. *AI Magazine*, 21(1):103–103, 2000.

[TOLR17] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. Change-preserving model repair. In *International Conference on Fundamental Approaches to Software Engineering*, pages 283–299. Springer, 2017.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

About the authors

Ludovico Iovino is Assistant Professor at the GSSI – Gran Sasso Science Institute, L'Aquila - in the Computer Science department. His interests include Model Driven Engineering (MDE), Model Transformations, Metamodel Evolution, code generation and software quality evaluation. Currently he is working on model-based artifacts and issues related to the metamodel evolution problem. He has been included in program commitees of numerous conferences and in the local organisation of the STAF 2015 and iCities 2018 conferences, he organised also the models and evolution workshop at MODELS 2018. He is part of different academic projects related to Model Repositories, model migration tools and Eclipse Plugins. Contact him at `ludovico.iovino@gssi.it`, or visit `http://www.ludovicoiovino.com`.

Angela Barriga is a PhD Candidate at Western Norway University of Applied Sciences. She has experience woring with machine learning, computer vision, gerontechnology and pervasive systems. Barriga's thesis is focused on model repair, specially on repairing using reinforcement learning. She has been part of the local organization of iFM 2019 and is involved in STAF 2020-2021. She is also part of the program committee of the third international workshop on gerontechnology. You can learn more about her at `https://angelabr.github.io/` or contact her at `abar@hvl.no`.

Adrian Rutle is a Full-time professor at Western Norway University of Applied Sciences. Adrian holds PhD in Computer Science from the University of Bergen, Norway. Rutle is professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen. Rutle's main interest is applying theoretical results from the field of model-driven software engineering to practical domains and has expertise in the development of modelling frameworks and domain-specific modelling languages. He also conducts research in the fields of modelling and simulation for robotics, eHealth, digital fabrication, smart systems and machine learning. Contact him at `adrian.rutle@hvl.no`

Rogardt Heldal is a professor of Software Engineering at the Western Norway University of Applied Sciences. Heldal holds an honours degree in Computer Science from Glasgow University, Scotland and a PhD in Computer Science from Chalmers University of Technology, Sweden. His research interests include requirements engineering, software processes, software modelling, software architecture, cyber-physical systems, machine learning, and empirical research. Many of his research projects are performed in collaboration with industry. Contact him at `rogardt.heldal@hvl.no`

PAPER C

A COMPARATIVE STUDY OF REINFORCEMENT LEARNING TECHNIQUES TO REPAIR MODELS

A. Barriga, L. Mandow, J.L. Perez de la Cruz, A. Rutle, R. Heldal and L. Iovino.

In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. 2020.

A comparative study of reinforcement learning techniques to repair models

Angela Barriga
abar@hvl.no
Western Norway University of Applied Science

Lawrence Mandow
lawrence@lcc.uma.es
Universidad de Málaga

José Luis Pérez de la Cruz
perez@lcc.uma.es
Universidad de Málaga

Adrian Rutle
aru@hvl.no
Western Norway University of Applied Science

Rogardt Heldal
rohe@hvl.no
Western Norway University of Applied Science

Ludovico Iovino
ludovico.iovino@gssi.it
Gran Sasso Science Institute, L'Aquila

ABSTRACT

In model-driven software engineering, models are used in all phases of the development process. These models may get broken due to various editions during the modeling process. To repair broken models we have developed PARMOREL, an extensible framework that uses reinforcement learning techniques. So far, we have used our version of the Markov Decision Process (MDP) adapted to the model repair problem and the Q-learning algorithm. In this paper, we revisit our MDP definition, addressing its weaknesses, and proposing a new one. After comparing the results of both MDPs using Q-Learning to repair a sample model, we proceed to compare the performance of Q-Learning with other reinforcement learning algorithms using the new MDP. We compare Q-Learning with four algorithms: Q(λ), Monte Carlo, SARSA and SARSA (λ), and perform a comparative study by repairing a set of broken models. Our results indicate that the new MDP definition and the Q(λ) algorithm can repair with faster performance.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; • **Theory of computation → Reinforcement learning**.

KEYWORDS

model repair, markov decision process, reinforcement learning

ACM Reference Format:
Angela Barriga, Lawrence Mandow, José Luis Pérez de la Cruz, Adrian Rutle, Rogardt Heldal, and Ludovico Iovino. 2020. A comparative study of reinforcement learning techniques to repair models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3417990.3421395

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3421395

1 INTRODUCTION

Models are becoming essential artifacts of modern software engineering processes [Whittle et al. 2014]. When performing modeling activities, the chances of breaking a model increase together with the size of development teams and the number of changes in software specifications, due to lack of communication, misunderstanding, mishandled collaborative projects, etc [Taentzer et al. 2017]. The correctness and accuracy of these models are of the utmost importance to correctly produce the systems they represent. However, it can be a time-consuming task to make sure that models are correct and have the required quality. Therefore, several approaches to automatic model repair have been proposed in the past decade [Macedo et al. 2013; Nassar et al. 2017; Ohrndorf et al. 2018].

In a previous work [Barriga et al. 2020] we presented our framework PARMOREL (Personalized and Automatic Repair of MOdels using REinforcement Learning) where users can personalize the repairing process. We proposed reinforcement learning (RL) [Sutton and Barto 2018] as a solution to allow both automatic and personalized model repair. RL consists of algorithms able to learn by themselves how to interact in an environment only needing a set of available actions and rewards for each of these actions. The structure of RL algorithms provides the necessary flexibility to adapt to different personalization settings and to improve performance after each execution.

In PARMOREL, the model repair problem is formulated as a Markov Decision Process (MDP) [Sutton and Barto 2018]. MDPs are defined in terms of a finite set of states and a finite set of actions. State transitions must depend only on the current state and the action chosen. In our formalization, the states are sets of errors in the model, and the set of actions is defined by the editing actions available provided by a modeling framework. The MDP is a theoretical framework and its concepts can be used to solve different problems. Each problem might require a different definition of the MDP concepts to be solved (a state can be a position in a maze, the score in a videogame, etc). However, the same problem could be solved with different definitions of the same MDP concepts. For

example, in the model repair problem so far we have defined the state space as each error contained in a model, but this is not the only state definition that could solve the problem.

Hence, in this paper, we want to explore an alternative definition of the MDP concepts for the model repair problem. We discuss our previous definition (MDP-A) and introduce a new one (MDP-B) that addresses the weak points of MDP-A. We analyze both MDPs strengths and weaknesses and compare them by repairing a corrupted model using the Q-Learning algorithm. The Q-learning algorithm provides several features that are useful to solve this problem in terms of reusability, structure, and decision making.

PARMOREL is not limited to the Q-learning algorithm, as it is built as an extensible framework and other algorithms for MDPs could be easily incorporated. Finally, we compare the results of Q-Learning and other RL algorithms in combination with the best MDP: Q(λ), Monte Carlo, SARSA, and SARSA (λ). We perform the comparative study repairing a dataset of models extracted from the dataset used in [Nguyen et al. 2019].

Structure of the paper. This paper is organised as follows: Section 2 presents the necessary background to understand the rest of the paper and introduces both MDP definitions, finishing with the results of repairing the same sample model with both MDPs and Q-Learning. Next, Section 3 presents different RL algorithms that can be an alternative to Q-Learning and finishes with the results of repairing the same dataset models with each algorithm. Then, we present threats to validity in Section 4, explore the related work in Section 5, and conclude the paper in Section 6.

2 FORMALIZING THE MODEL REPAIR PROBLEM AS A MARKOV DECISION PROCESS

MDPs are mathematical models used to solve sequential decision-making problems [Mundhenk et al. 2000]. At specified points in time, a decision agent observes the state of a system and chooses an action. The action choice and the state make the system transition to a new state at a subsequent discrete point in time. The agent receives a reward signal at each transition. The goal of the MDP is to find a policy (i.e. a mapping from states to actions) that maximizes the rewards accumulated over time.

As long as one commits to this behaviour, these concepts of state, action, and reward are abstract and must be defined in order to apply MDP to solve a specific problem.

In this section, we present two different formalizations of the model repair problem using MDPs: our previous approach, from now on MDP-A and our new one, MDP-B. In each approach we define and discuss the following concepts:

(1) State space: set of states, observable situations, that can happen in a system. Every system has an initial state (how it starts) and a final state. The goal of the MDP is to find a policy of actions that takes the system from the initial to the final state while maximizing the accumulated reward, i.e. an optimal policy. It is important to differentiate between a state and the actual system. A state is what is observable by the agent and it might not contain all details about the system, because they might not be necessary or available to the agent. For example, given the model repair problem, our system would be the model itself, but the state is what we consider important in the model to solve the problem, for instance, the errors present in the model but not its whole structure.
(2) Action space: the set of actions that can modify the system, leading to new states.
(3) Reward: a numerical value that tells the agent how good is the action it applies. The reward is local and immediate, so it does not reflect future consequences.

Before diving into each approach, it is necessary to clarify some concepts necessary to follow further explanations:

Step: a step corresponds with the application of one action in the system.

Episode: each episode corresponds with one iteration in which the algorithm has successfully repaired the model using the available actions. We consider an episode ends when the final state is reached. The number of episodes is finite, we define a maximum number of episodes for the algorithm to run. A good number of episodes is when the algorithm has sufficient time to find the optimal policy of actions.

Customizable rewards: rewards can be adapted to align with user preferences to personalize the repair result. Since rewards indicate how good local actions are, the only requirement for user preferences is that they can be quantified (e.g., preserve the original model structure by minimizing the model distance metric or boost quality characteristics by optimising quality metrics). Since PARMOREL is extensible, users can plug different tools to obtain these rewards. In this paper, we work with a quality metrics tool [Iovino et al. 2020] inspired by the quality model proposed in [Basciani et al. 2016]. The rewards obtained by using quality metrics correspond with a positive float number. From now on we refer to this component of PARMOREL as the metrics tool.

Q-Learning: an RL algorithm that solves MDP problems. Knowledge acquired is stored in a table structure called Q-Table. This table stores pairs of states and actions together with a Q-value. The Q-value is calculated using the rewards and it indicates how good each pair is. The Q-value is obtained with repeated calculations based on the Bellman Equation [Bellman 2013] (see Equation 1), telling that the maximum future reward is the reward r the agent received for entering the current state s_t with some action a_t plus the maximum future reward for the next state s_{t+1} and action a_{t+1} reduced by a discount factor γ. This allows inferring the value of the current (s_t, a_t) pair based on the estimation of the next one s_{t+1}, which can be used to calculate an optimal policy to select actions. The factor α provides the learning rate, which determines how much new experience affects the Q-values. One of the variables used to calculate the Q-value, is the maximum weight stored in the Q-table for the next error to repair ($\max_{a'} Q(s_{t+1}, a')$). This allows us to measure the consequences of applying a certain action in the model (e.g., if applying an action creates a new unknown error this action will be punished, getting a lower weight). At the end of the execution, pairs with the highest Q-value will conform to the policy to solve the problem. Our algorithm is epsilon-greedy (ε-greedy): it avoids local optima using an exploration-exploitation trade-off

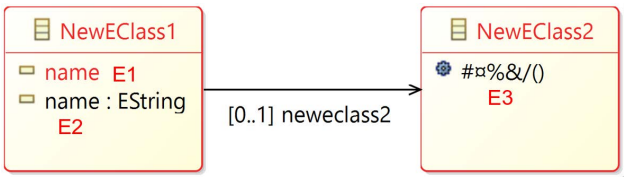


Figure 1: Sample model with 3 errors

by exploring (i.e. choosing a random action) with probability ε, and exploiting (i.e. choosing the action with highest Q-value) the remainder of the time. We work with an ε of 0.3. Regarding other parameters, discount factor (γ), and learning rate (α), we use 1.0 for both of them. Details of Q-Learning can be seen in Algorithm 1, which has been adapted from [Sutton and Barto 2018] (cf. chapter 6).

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s,a)) \quad (1)$$

Algorithm 1 Q-Learning

```
1:  Initialize Q-Table
2:  for each episode do
3:      s ← initial state s_0
4:      while errors in model != ∅ do
5:          Get state s
6:          Select action a with ε-greedy policy for s
7:          s_{t+1} ← a applied in s
8:          Q(s,a) = Q(s,a) + α(r + γ max_{a'} Q(s_{t+1}, a') − Q(s,a))
9:          t ← t + 1
10:         s ← s_{t+1}
```

We use the sample model in Figure 1 to exemplify and compare each MDP definition. It has two classifiers, one reference, two attributes and, an operation. The following errors are present in the model:

(1) **E1:** Attribute without a type.
(2) **E2:** Two attributes with repeated names.
(3) **E3:** Operation with a not well formed name.

2.1 MDP-A definition

We start by explaining the concepts of MDP-A, the MDP definition we have used in PARMOREL so far.

State space: the state space is defined by the list of errors present in the model. A state corresponds with a single error in the model, so errors are repaired one by one, following their order in the list obtained from the modeling framework. The final state is one without errors, i.e. it corresponds to a repaired model. In order to obtain the errors, we rely on the modeling framework in which the models are produced (e.g., the Eclipse Modeling Framework, EMF [Steinberg et al. 2008]). Some examples of these errors are syntactical errors of conformance with the Ecore metamodel: the opposite must be a feature of the reference's type, two or more features with the same name, etc. An example of a state would be Fig. 1's initial state: [E1].

Action space: the set of editing actions obtained, as with the errors, from the modeling framework in which the models are produced. For each state, actions are filtered, so that only actions capable of repairing at least one error in the state are considered. Some examples of these actions are: delete, setName, setType, setContainment, etc.

Reward: in every step, we reward an applied action by using an external tool used to measure the quality of the model in terms of quality metrics, model distance, etc.. Then, when all episodes finish, PARMOREL picks the sequence of actions with the highest accumulated rewards and provides an extra reward to these actions. With this, we reward each action not only for how good they are individually but also for belonging to the best repairing sequence.

2.1.1 MDP-A strengths and weaknesses. We have used this approach in enough experiments to identify a series of points that could be improved. Since we give substantial rewards at every step, fewer episodes are necessary to find good enough repair sequences. In this approach, we do not give the algorithm enough time to converge to an optimal solution by itself, since we consider every solution found, even if it is not optimal. At the end of the episodes, PARMOREL compares all sequences found and picks the one with higher accumulated rewards. This reduces the execution time, however, there might be better solutions that the algorithm does not find due to not having enough episodes to do so. Also, in an orthodox Q-Learning approach, the algorithm should converge to the optimal solution, without comparing the found sequences.

Although it is not necessary to provide all details of the model in the state, there might be situations in which our current state definition might not provide enough information about the model. For instance, there might be two models defined by the same state space because they contain the same errors, but they might be different in terms of structure and they would need different repair solutions. For example, we could have a small model with all errors concentrated in one class and another big model with the same type of errors scattered in its structure. Although the initial state would be the same for both models their optimal repair sequence might be very different. This might lead to having stochastic rewards, this is different rewards for the same state.

Regarding Q-Learning, the Q-table is populated before algorithm execution. This means we process all errors existing in the model and actions to repair each of them and create a respective entry in the Q-table. Errors are stored in the Q-table individually. An advantage of this approach is that it makes the Q-table easier to reuse since the same errors can appear in many different models and we are storing information about them individually. However, this way the algorithm learns to solve errors one by one, while there might be scenarios where it could be able to repair several ones at the same time, which would be more efficient. Also, by forcing individual repair we might be forcing a repair order that might not be the best one.

2.2 MDP-B definition

To address the weak points of our previous MDP approach, we propose an alternative formalization, exploring different, more

elaborate definitions of the state space and reward function. The definition of action space remains unchanged.

State space: Each state is defined by a set of model errors together with an array containing the number of classes (NC), the total number of attributes, and operations (NA) and the total number of references (NR) in the model. All these parameters are positive integers. This array is updated at the beginning of each episode with the current errors present in the model and the values of NC, NA, and NR. A final state has an empty set of errors, i.e, it stands for a repaired model. The final state is independent of the NC, NA and, NR values. Some examples of these errors are: "the opposite must be a feature of the reference's type", "two or more features with the same name", etc. An example of a state would be the initial state in Fig 1: {E1, E2, E3} [2, 3, 1].

Reward: in every non-final step the reward will be 0. When the final state is reached, the reward will be given by an external tool used to measure the quality of the repaired model in terms of quality metrics, model distance, etc.

By giving the reward once the final state is reached, the algorithm will need more episodes to learn how to repair the model but in exchange, it will always find the optimal repair sequence.

2.2.1 MDP-B strengths and weaknesses. MDP-B offers several advantages over MDP-A. By including NC, NA and, NR in the states, we give the algorithm some information about the model structure, making a more complete observation. Returning to the example mentioned in the MDP-A, now if we have two models with the same errors but very different structures, the differences will be reflected in the states.

Now the Q-table is populated dynamically when the algorithm finds new states. This helps to reduce the longer learning time caused by a higher number of episodes. In this MDP approach, we store in the Q-table the complete set of errors present in the model, instead of individual errors. Now the algorithm can learn which is the optimal repair sequence without forcing it to repair the errors one by one, as in the MDP-A. However, this might make the Q-table harder to reuse. This problem could be mitigated by adapting our transfer learning approach [Barriga et al. 2020] to work with this new Q-table.

2.3 Comparing both MDP formalizations

We now proceed to compare the MDP approaches presented in the previous section. We compare both approaches in terms of learning time and the number of episodes required to find the optimal solution. We use Q-Learning as our learning algorithm and EMF as our modeling framework. As user preference, we decide to boost the *maintainability* of the models [Iovino et al. 2020]. The *maintainability* quality metric considered in this paper has been defined according to the definition given in [Genero and Piattini 2001] which is based on some of the metrics shown in Table 1 as follows,

$$Maintainability = \left(\frac{NC + NA + NR + DIT_{Max} + Fanout_{Max}}{5} \right) \quad (2)$$

Metric	Acronym
Number of classifiers	NC
Number of references	NR
Number of attributes	NA
Max generalization hierarchical level	DITmax
Max reference sibling	FANOUTmax

Table 1: Excerpt of the metrics considered in the evaluation

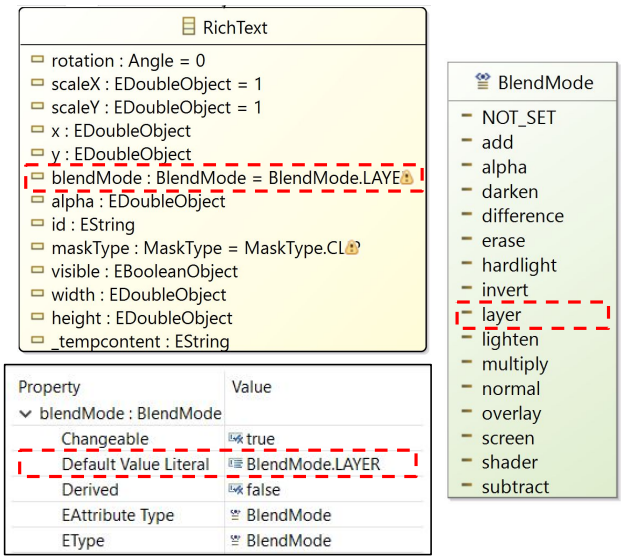


Figure 2: Error 38 indentified in fxg.ecore

According to the considered definition of *maintainability* the lower the values the better. Since Q-Learning maximizes the reward, PARMOREL translates maintainability values to the negative so that the lower they are the higher is the reward they provide.

To test both MDPs, we use a sample model extracted randomly from our dataset of models (more details in the next section). This model needs to be repaired because it contains syntactic errors that violate certain constraints of the Ecore metamodeling language [Steinberg et al. 2008].

Specifically, the sample model *fxg.ecore* contains eight instances of error 38: "Invalid specified literal". In Fig. 2 we show one of the instances of this error in *fxg.ecore*. The model contains 52 classes, 37 references, and 267 attributes and operations. Error 38 "Invalid specified literal" is a warning saying that the default value specified is not coherent with the literals specified in the enumeration. Indeed in the case in Fig. 2 the default value specified for the attribute blendMode is *BlendMode.LAYER*, when the corresponding literal on the enumeration, set as datatype, is *layer*. Possible solutions are to modify the literal value in the attribute with any of the possible literals, modify the default in the datatype enumeration, remove the faulty attribute, etc.

We run both approaches until the maximum Q-value of the initial state ($max_a Q(s_0, a)$) remains unchanged for 1000 steps. With this

	Time (s)	Episodes	Improvement
MDP-A	231.34	548	-
MDP-B	88.29	217	61.84%

Table 2: Comparison of MDP approaches

Code	Message
Error 13	The opposite must be a feature of the reference's type
Error 29	Two or more Classifier with the same name
Error 32	Two or more feature with the same name
Error 38	Invalid specified literal

Table 3: Errors present in the selected models

number of steps, we give the algorithm enough time to converge to the best solution.

First, we proceed to repair fxg.ecore using PARMOREL with the MDP-A approach. The process took 231.34s and converged in 548 episodes.

Secondly, we repeat the experiment this time using PARMOREL with the MDP-B approach. In this occasion, the process took a total of 88.29s and converged after 217 episodes.

For both MDPs, the resulting model produced with the best repairing policy found has a maintainability score of 72.2. This sequence includes actions that modify the literal value and remove some of the faulty attributes. It is interesting to mention that, for both approaches, between 70-75% of the learning time is dedicated to calculate the quality characteristics of the model. So the time dedicated only to learn to repair using Q-Learning constitutes only a 25-30% of the total.

The best result in terms of learning time and the number of episodes is obtained when executing PARMOREL with the MDP-B. Table 2 displays a summary of the results. The total learning time is reduced by 61.84% with respect to the MDP-A. As explained in the previous section, one of the weakest points of MDP-A was that it was not specifically designed to converge to the best solution but to find a range of different solutions. This, together with the population of the Q-table before the algorithm execution, naturally leads to a slower convergence. In contrast, MDP-B is designed to converge to the optimal solution, and its dynamic population of the Q-table reduces drastically the learning time. Hence, given these results, we consider the MDP-B a better approach to tackle the model repair problem with RL than our previous MDP-A definition.

3 ALTERNATIVE REINFORCEMENT LEARNING ALGORITHMS

Next, in order to assess if there is an alternative to Q-Learning that provides faster performance when repairing models, we apply the MDP-B using different RL algorithms to repair the same dataset.

To conduct a preliminary evaluation, we use a representative sample of 12 models from the dataset used in [Nguyen et al. 2019], filtered in order to get only corrupted Ecore models. All errors present in these models are syntactic errors that violate certain constraints of the Ecore metamodeling language [Steinberg et al. 2008] (e.g., the opposite of the opposite of a reference must be the reference itself, classifiers must have different names, etc). Each subject model contains between 1 and 8 errors (details of the errors can be seen in Table 3). Regarding the number of elements, each model has between 10 and 352, counting the number of classes, attributes, references, and operations.

Given the stochastic nature of the ε-greedy policy used by the learning algorithms, we average the performance data (learning time and number of episodes) for each model over 10 repair agents with different random seeds. By changing the seeds, each agent will pick different actions when choosing randomly. All results displayed in this section are averaged data. The user preference, as in Section 2.3 is to boost the maintainability of the models using Equation 2.

We begin by obtaining the performance of Q-Learning with MDP-B repairing the dataset. Later, we use the results as a benchmark to compare Q-Learning with other RL algorithms.

To run these experiments, we use the same configuration as in Section 2: an ε of 0.3, and both γ and α of 1.0 (we keep these values constant since, according to our testing, they provide the best results regardless the algorithm). Also, to find the number of episodes at which each algorithm converges, we run the algorithms with the same stopping criteria presented earlier: each execution will stop once $max_a Q(s_0, a)$ (the maximum Q-value of the initial state) remains unchanged for 1000 steps. The values in the Q-table are initialized to -500.0 so that it contains a value lower than the worst maintainability found in the models. We compare our Q-Learning results with the following algorithms: Q(λ), Monte Carlo, SARSA and, SARSA(λ). We introduce each algorithm in the following subsections.

3.1 Q(λ)

This algorithm [Sutton and Barto 2018] is a mixture of the ideas behind temporal difference methods (such as Q-learning, where each value is updated according to the immediate step), and those of Monte Carlo methods (more details in the following subsection). The algorithm uses a technique called *eligibility traces* to back-propagate the values and rewards received (as in temporal differences), but it does so not only to the immediately preceding state (or pair of action-state), but to all preceding states of the current episode. The idea is that this propagation decays in intensity the further a state is in the past, see eligibility e(s,a) in Algorithm 2. This decayed propagation can lead to a speed up in the algorithm's convergence, especially in sparse reward models, like MDP-B, which provides reward only at the end of each episode. The propagation decay is controlled with a parameter λ, such that if its value is 0 the algorithm would behave like Q-Learning and if it is 1 it would behave like a pure Monte Carlo method. In practice, the speed of convergence as a function of the value of λ(between 0 and 1) generally has a U-shape. Therefore, the optimal convergence is usually achieved with an intermediate value of λ, which needs to be determined experimentally. According to our experiments, we get the best results by giving λ a value of 0.7. Lower or higher values lead to results of lower quality and slower convergence.

The use of eligibility traces is a general idea that can be executed with different implementations. We follow the Q(λ) algorithm since it can be implemented on top of Q-Learning, modifying some parts of the latter. The new parts added to Q-Learning are depicted in

Algorithm 2. This pseudocode is adapted from the one presented in chapter 12 in [Sutton and Barto 2018].

Algorithm 2 Q(λ)

Initialize Q-Table
for each episode **do**
    Initialize eligibility table e (default value 0)
    Initialize sae as an empty list of state-action pairs
    $s \leftarrow$ *initial state* s_0
    while errors in model != ∅ **do**
        Get state s
        Select best action a with ϵ-greedy policy for s
        if a is selected randomly **then**
            reset eligibility to 0
            reset sae as an empty list
        $s_{t+1} \leftarrow$ *a applied in s*
        Add (s,a) to sae list
        $e(s,a) \leftarrow e(s,a) + 1$
        $\delta_t = r + \gamma max_{a'} Q(s_{t+1}, a') - Q(s,a)$
        for each s,a in sae **do**
            $Q(s,a) = Q(s,a) + \alpha\delta e(s,a)$
            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
        $t \leftarrow t + 1$
        $s \leftarrow s_{t+1}$

3.2 Monte Carlo methods

Monte Carlo methods are ways of solving RL problems based on averaging sample returns [Sutton and Barto 2018]. As in other RL algorithms, in Monte Carlo, an agent learns about the states and rewards when it interacts with the environment. In this method the agent generates experienced samples and then based on the average return, a value is calculated for a state or pair of state-action. The main difference of this algorithm is that the agent learns by sampling experience. Unlike Q-learning, Monte Carlo learns directly from episodes of experience without any prior knowledge of MDP transitions.

One weakness is that it can only be applied to episodic MDPs. The reason is that each episode has to terminate before the algorithm can calculate any returns. In the model repair scenario we tackle in this paper this is not a problem since we obtain the maintainability of the produced model only when an episode ends. Also, in Monte Carlo, there is no guarantee to visit all the possible states and the sampling process can lead to high variance of results.

To execute this algorithm, we proceed to run the Q(λ) algorithm (see Algorithm 2) with λ set to 1.0. This way, the algorithm behaves as a Monte Carlo method without requiring further modification.

3.3 SARSA

SARSA is a temporal-difference method that differs from Q-Learning in that the target value for the learning update rule is $Q(s_{t+1}, a_{t+1}$ instead of $max_{a'} Q(s_{t+1}, a')$, where a_{t+1} is the action actually chosen by the ϵ-greedy learning strategy during the episode.

SARSA is strongly dependant on the value of ε, to the extent that convergence will only happen when ε decreases to 0. So, unlike the other algorithms where ε was constant with a value of 0.3, for executing SARSA we need to decrease ε during the execution. According to our experiments, starting with a value of 0.3 and decreasing it by multiplying it by 0.995 at the end of each episode provides the best results we could find. If ε decreases too fast the converge will be slower, and if it decreases too slow it might not find the best solution. Details about SARSA can be found in Algorithm 3, adapted from chapter 6 in [Sutton and Barto 2018].

Algorithm 3 SARSA

Initialize Q-Table
for each episode **do**
    $s \leftarrow$ *initial state* s_0
    Select action a with ϵ-greedy policy for s
    while errors in model != ∅ **do**
        Get state s
        $s_{t+1} \leftarrow$ *a applied in s*
        Select a_{t+1} *with ϵ-greedy policy for* s_{t+1}
        $Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s_{t+1}, a_{t+1}) - Q(s,a))$
        $t \leftarrow t + 1$
        $s \leftarrow s_{t+1}$
        $a \leftarrow a_{t+1}$

3.4 SARSA(λ)

SARSA(λ) is the combination of the SARSA algorithm with eligibility traces implemented on top of it, just as happened with Q(λ) and Q-Learning.

The main difference with Q(λ) is the same as between Q-Learning and SARSA, the way a_{t+1} is selected. Due to the eligibility traces behavior, SARSA(λ) is more independent of ε than regular SARSA, so it is not necessary to decrease ε during the execution. Details about SARSA(λ) can be found in the pseudocode in Algorithm 4, adapted from the code presented in chapter 12 in [Sutton and Barto 2018].

3.5 Results of the algorithmic comparison

The results obtained from each algorithm are displayed in tables 4 and 5. Table 4 displays how long it takes in average for each algorithm to learn to repair each model in the dataset (in seconds) and how many episodes each algorithm needs to converge to the best solution. Table 5 shows the total time each algorithm needs to learn to repair all the models in the dataset and the percentage of improvement each algorithm presents with retrospect to Q-Learning. If the improvement is a negative number it means that the algorithm performs poorer than Q-Learning. We take Q-Learning as the reference since it was the algorithm used in PARMOREL so far.

The algorithm that presents the best performance, both in time and number of episodes is Q(λ), completing the repair 3.53% faster than Q-Learning. The rest of the algorithms, Monte Carlo, SARSA(λ) and SARSA, perform worse than Q-Learning, being Monte Carlo the one with a better performance from this group, with -3.8% and SARSA the worse by a big difference, performing a 61% worse than Q-Learning.

	Q-Learning		Q(λ)		Monte Carlo		SARSA		SARSA(λ)	
Model	**Time (s)**	**Episodes**	**Time (s)**	**Episodes**	**Time (s)**	**Episodes**	**Time (s)**	**Episodes**	**Time (s)**	**Episodes**
activityDiagram	93.39	334	89.87	320	93.42	335	449.97	1367	102.31	355
aggregator_0.9.0	295.32	697	263.83	697	297.23	697	481.14	1215	298.15	699
backbone	172.12	545	168.51	532	177.69	581	378.61	1210	206.1	716
bpmn20	559.9	1031	526.57	1015	531.79	1031	514.99	1030	556.3	1031
BusinessDomainDsl	205.92	694	202.78	672	208.98	694	389.38	1215	208.14	697
car	298.33	1003	295.1	1001	298.17	1003	301.17	1009	299.59	1005
chess	308.1	1043	306.87	1033	307.24	1037	309.28	1052	396.92	1033
family	298.58	1007	296.51	1001	299.76	1007	326.18	1109	297.72	1007
fxg	88.29	217	82.17	206	178.75	464	470.47	1195	130.71	341
General	332.08	1079	324.17	1033	338.56	1085	392.3	1121	323.96	1036
glucose	97.14	353	96.84	352	99.17	369	264.89	870	97.81	382
GSML	182.39	595	174.85	590	212.28	673	441.7	1373	176.04	595

Table 4: Algorithms comparison in episodes and time

	Q-Learning	Q(λ)	Monte Carlo	SARSA	SARSA(λ)
Total time (s)	2931.56	2828.07	3093.75	4720.08	3093.75
Improvement	-	3.53%	-3,8%	-61%	-5.53%

Table 5: Algorithms compared with Q-Learning

Algorithm 4 SARSA(λ)

Initialize Q-Table
for each episode **do**
  Initialize eligibility table e (default value 0)
  Initialize sae as an empty list of state-action pairs
  $s \leftarrow$ *initial state* s_0
  Select action a with ϵ*-greedy policy for s*
  while errors in model != ∅ **do**
    Get state s
    $s_{t+1} \leftarrow$ *a applied in s*
    Select a_{t+1} *with* ϵ*-greedy policy for* s_{t+1}
    if a_{t+1} is selected randomly **then**
      reset eligibility to 0
      reset sae as an empty list
    $s_{t+1} \leftarrow$ *a applied in s*
    Add (s,a) to sae list
    $e(s,a) \leftarrow e(s,a) + 1$
    $\delta_t = r + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$
    for each (s,a) in sae **do**
      $Q(s,a) = Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $t \leftarrow t + 1$
    $s \leftarrow s_{t+1}$
    $a \leftarrow a_{t+1}$

These results are an indicator of the potential of Q(λ) for the model repair problem, while Monte Carlo, SARSA and SARSA(λ) might not provide the best solutions.

Additionally, Table 6 displays the maintainability of each model in the dataset before and after repair. The repaired models present better maintainability than their broken versions.

	Maintainability	
Model	**Before repair**	**After repair**
activityDiagram	7.6	6.8
aggregator_0.9.0	52.2	51.8
backbone	7.6	7.2
bpmn20	121.6	121.4
BusinessDomainDsl	15.4	15.0
car	2.8	2.6
chess	3.0	2.8
family	3.8	3.2
fxg	73.2	72.2
General	25.8	25.6
glucose	10.4	9.8
GSML	6.4	4.6

Table 6: Models maintainability before and after repair

4 THREATS TO VALIDITY

In this section, we comment on the threats to validity of our research, following the guidelines from [Wohlin et al. 2012].

Internal threats. Among the internal threats, we have the quality evaluation process since the quality model is user-defined. Quality aspects are often based on the modeler's experience and mistakes in these quality models' definitions may impact the results. Including experts in the quality definition process can mitigate this aspect, e.g. by including definitions adopted from the literature. In this direction, the formula we used for maintainability is based on the literature.

External threats. A potential external threat to the validity of our evaluation is the dataset used for the experiments. We have selected corrupted models resulting in a dataset of 12 models, which may be considered small, however, this threat may be mitigated with

the heterogeneity of the sources; these models have been retrieved from the dataset used in [Nguyen et al. 2019] which comes from different Github repositories and hence from different modelers.

RL is a wide field with multiple algorithms. We have selected a set of four of them based on their suitability to solve the model repair problem and the proximity to the current implementation (Q-Learning). We consider they are a representative set to indicate which algorithms should be explored further in our research.

Although our implementation is tied to EMF and Ecore models, PARMOREL is built as an Eclipse plugin, so it is possible to use other modelling frameworks—through the implementation of a series of interfaces.

5 RELATED WORK

Model repair is a research field that has drawn the interest of many researchers to formulate approaches and build tools to repair broken models. Despite the variety, we could not find in the literature any research that applies RL to model repair. Hence, in this section we explore literature that uses other types of techniques to repair models.

Kretschmer et al. introduce in [Kretschmer et al. 2018] an approach for discovering and validating values for repairing inconsistencies automatically. Values are found by using a validation tree to reduce the state space size. Trees tend to lead to the same solutions once and again due to their exploitation nature (probing a limited region of the search space). Differently, RL algorithms include both exploitation and exploration (randomly exploring a much larger portion of the search space with the hope of finding other promising solutions that would not be selected normally), allowing to find new and, sometimes more optimal solutions for a given problem.

Also tree-powered, Model/Analyzer [Reder and Egyed 2012] is a tool that, by using the syntactic structure of constraints, determines which specific parts of a model must be checked and repaired. The user is expected to select a specific violation to be repaired but does not support user customization., unlike what we do with RL rewards.

Puissant et al. propose a tool called Badger based on an artificial intelligence technique called automated planning [Puissant et al. 2015]. Badger generates sequences that lead from an initial state to a defined goal. It has a set of repaired operations to which users can assign costs and weights to decide its priority. Badger generates a set of plans, each plan being a possible way to repair one error. This makes it difficult for the user to decide which action to apply without knowing how it affects the rest of the model. We prefer to generate alternative sequences to repair the whole model since some repair actions can modify the model drastically.

It is worth mentioning search-based and genetic algorithm-based approaches since, although they have not been applied yet to model repair, they are possible competitors to RL. These techniques have shown promising results dealing with model transformations and evolution scenarios, for example in [Kessentini et al. 2017] authors use a search-based algorithm for model change detection. These algorithms deal efficiently with large state spaces, however they cannot learn from previous tasks nor improve their performance. While RL is, in the beginning, less efficient in large state spaces, it can compensate with its learning capability. In the beginning, performance might be poor, but with time repairing becomes straightforward.

Lastly, another search-based approach is presented by Moghadam et al. in [Moghadam and Ó Cinnéide 2011]. In this work, authors present Code-Imp, a tool for refactoring Java programs based on quality metrics that achieves promising results at code-level by using hill-climbing algorithms [Selman and Gomes 2006]. These algorithms are interesting to find a local optimum solution but they do not assure to find the best possible solution in the search space (the global optimum). By using RL we assure to find the global optimum: the sequence of repairing actions that maximize the selected quality characteristics the most.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a comparative study of different RL techniques to solve the model repair problem in our tool PARMOREL. First, we proposed a new MDP definition to address the weaknesses of our previous MDP and compared both approaches with the Q-Learning algorithm in a sample model. Then, using the new MDP definition, we compared the performance of Q-Learning with other RL algorithms, namely, Q(λ), Monte Carlo, SARSA and, SARSA(λ). We applied each algorithm to repair a dataset of models extracted from the dataset used in [Nguyen et al. 2019]. Although our results are preliminary, we consider them an indicator of the potential of Q(λ) for repairing models.

In the future, we would like to make further research about Q(λ) and other RL techniques. Another interesting line of future research would be to explore a multi-objective approach to model repair [Mandow and Pérez-de-la Cruz 2018], considering and combining different performance metrics.

Additionally, we would like to perform a comparative study with the automatic repairing tools presented in Section 5, paying especial attention to search-based and automated planning approaches. We will also include less automatic approaches, like those that are search-based. Lastly, in this direction, we will work on optimizing the repair with a focus on achieving state-of-the-art time.

Also, we plan to extend this comparative study with a wider dataset of domain models and errors, with the help of modelers that may attest if the repaired sequence offers a better quality of the repaired domain model. In particular, we plan to test the presented approach with a bigger dataset of domain models coming from GitHub repositories, in order to validate the approach with real examples. Additionally, we plan to create a benchmark with the mentioned dataset, with which we will compare PARMOREL to other existent model repair approaches.

ACKNOWLEDGEMENTS

L. Mandow and J.L. Pérez de la Cruz are funded by the Spanish Government, Agencia Estatal de Investigación (AEI) and European Union, Fondo Europeo de Desarrollo Regional (FEDER), grant TIN2016-80774-R (AEI/FEDER, UE).

REFERENCES

Angela Barriga, Adrian Rutle, and Heldal Rogardt. 2020. Improving model repair through experience sharing. *Journal of Object Technology* 19, 2 (July 2020), 13:1–21. https://doi.org/10.5381/jot.2020.19.2.a13

Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2016. A customizable approach for the automated quality assessment of modelling artifacts. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 88–93.

Richard Bellman. 2013. *Dynamic programming*. Courier Corporation.

Marcela Genero and Mario Piattini. 2001. 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*.

Ludovico Iovino, Angela Barriga, Adrian Rutle, and Heldal Rogardt. 2020. Model Repair with Quality-Based Reinforcement Learning. *Journal of Object Technology* 19, 2 (July 2020), 17:1–21. https://doi.org/10.5381/jot.2020.19.2.a17

Marouane Kessentini, Usman Mansoor, Manuel Wimmer, Ali Ouni, and Kalyanmoy Deb. 2017. Search-based detection of model level changes. *Empirical Software Engineering* 22, 2 (2017), 670–715.

Roland Kretschmer, Djamel Eddine Khelladi, and Alexander Egyed. 2018. An automated and instant discovery of concrete repairs for model inconsistencies. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 298–299.

Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. 2013. Model repair and transformation with Echo. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 694–697.

Lawrence Mandow and José-Luis Pérez-de-la Cruz. 2018. Pruning Dominated Policies in Multiobjective Pareto Q-Learning. In *Conference of the Spanish Association for Artificial Intelligence*. Springer, 240–250.

Iman Hemati Moghadam and Mel Ó Cinnéide. 2011. Code-Imp: a tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*. 41–44.

Martin Mundhenk, Judy Goldsmith, Christopher Lusena, and Eric Allender. 2000. Complexity of finite-horizon Markov decision process problems. *Journal of the ACM (JACM)* 47, 4 (2000), 681–720.

Nebras Nassar, Hendrik Radke, and Thorsten Arendt. 2017. Rule-Based Repair of EMF Models: An Automated Interactive Approach. In *International Conference on Theory and Practice of Model Transformations*. Springer, 171–181.

Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. 2019. 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, 272–282.

Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2018. ReVision: a tool for history-based model repair recommendations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 105–108.

Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. 2015. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling* 14, 1 (2015), 461–481.

Alexander Reder and Alexander Egyed. 2012. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 220–229.

Bart Selman and Carla P Gomes. 2006. Hill-climbing Search. *Encyclopedia of cognitive science* (2006).

Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. 2017. Change-preserving model repair. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 283–299.

Jon Whittle, John Hutchinson, and Mark Rouncefield. 2014. The state of practice in model-driven engineering. *IEEE software* 31, 3 (2014), 79–85.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

PAPER D

AN EXTENSIBLE FRAMEWORK FOR CUSTOMIZABLE MODEL REPAIR

A. Barriga, R. Heldal, L. Iovino, M. Marthinsen and A. Rutle.

In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. 2020.

An extensible framework for customizable model repair

Angela Barriga
abar@hvl.no
Western Norway University of Applied Science

Rogardt Heldal
rohe@hvl.no
Western Norway University of Applied Science

Ludovico Iovino
ludovico.iovino@gssi.it
Gran Sasso Science Ististute, L'Aquila

Magnus Marthinsen
magmar@online.no
Western Norway University of Applied Science

Adrian Rutle
aru@hvl.no
Western Norway University of Applied Science

ABSTRACT

In model-driven software engineering, models are used in all phases of the development process. These models may get broken due to various editions during the modeling process. There are a number of existing tools that reduce the burden of manually dealing with correctness issues in models, however, most of these tools do not prioritize customization to follow user requirements nor allow the extension of their components to adapt to different model types. In this paper, we present an extensible model repair framework which enables users to deal with different types of models and to add their own repair preferences to customize the results. The framework uses customizable learning algorithms to automatically find the best sequence of actions for repairing a broken model according to the user preferences. As an example, we customize the framework by including as a preference a model distance metric, which allows the user to choose a more or less conservative repair. Then, we evaluate how this preference extension affects the results of the repair by comparing different distance metric calculations. Our experiment proves that extending the framework makes it more precise and produces models with better quality characteristics.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; • **Theory of computation → Reinforcement learning**.

KEYWORDS

model repair, reinforcement learning, model distance, quality evaluation

ACM Reference Format:
Angela Barriga, Rogardt Heldal, Ludovico Iovino, Magnus Marthinsen, and Adrian Rutle. 2020. An extensible framework for customizable model repair. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410957

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410957

1 INTRODUCTION

Models are central objects of the processes of modern software engineering [Whittle et al. 2014]. When conducting modeling activities, modelers can introduce errors of different nature in the models (syntactic errors, duplicates, bad smells [Bettini et al. 2019], antipatterns [Strittmatter et al. 2016], etc), making them corrupted. The chances of corrupting a model increase along with the size of development teams and amount of changes in software requirements due to [Taentzer et al. 2017] lack of coordination, misunderstanding, mishandled collaborative projects, etc.

The reliability and accuracy of these models is of utmost importance to correctly produce the systems they represent. But ensuring that the models are accurate, have the required quality and remain true to the original model structure can be a time-consuming task. A variety of solutions to automatic model repair have therefore been suggested over the last decades, tackling repair of corrupted models from different perspectives and applied to different kinds of models: [Macedo et al. 2013; Nassar et al. 2017; Ohrndorf et al. 2018].

Additionally, there are multiple, possible repair solutions that a modeler could choose while there might not exist an objectively best solution to satisfy all modelers. Consequently, the modeling community has developed over the years a series of metrics and characteristics that can be used to get an unbiased measure of how good is a model.

For example, the literature has already highlighted how important it is to preserve the original model structure when repairing in order to minimize undesired side-effects in the repaired model [Khelladi et al. 2019; Taentzer et al. 2017]. An effective tool to measure the preservation of the original model structure is the calculation of the distance between the original and the repaired model [Addazi et al. 2016; Kehrer et al. 2011; Syriani et al. 2019]. However, within the approaches developed by the community to automatically repair models, minimizing the model distance with respect to the original has not been the main focus of the repairing algorithms.

Another example is the use of characteristics specifically conceived to measure the quality of models and other modeling artifacts, like analyzability, adaptability, or understandability [Basciani et al. 2019; López-Fernández et al. 2014]. Even though quality characteristics have been extensively studied in the literature [Boehm et al. 1976; Dromey 1995; Ortega et al. 2003], the quality of the automatically repaired models has not been the main focus of the repairing algorithms.

We can conclude that there are multiple approaches both to solve the model repair problem in different kinds of models and to measure the quality of the produced models. However, if a modeler wants to apply and compare different approaches, she will need to download and use several tools or to implement one herself, since there is no unified way to perform this process at the moment.

In our previous work [Barriga et al. 2019, 2020; Iovino et al. 2020], we presented our research of model repair and reinforcement learning (RL) [Thrun and Littman 2000] through PARMOREL. PARMOREL is an approach that provides personalized and automatic repair of software models using RL algorithms. It finds a sequence of repairing actions according to preferences introduced by the user. So far, we conceived PARMOREL as a tool to provide repair of models only providing personalization of the users preferences. However, due to the flexibility RL provides to our approach, we want to transform PARMOREL from a rigid tool to framework that users can fully customize to their needs.

Hence, in this paper, we present our new version of PARMOREL, an extensible model repair framework which allows customization of results, type of models and learning.

In this paper, we explore PARMOREL's suite of customizable modules, which allows users to choose and add their own repair preferences, to work with different modeling frameworks and with different learning algorithms.

Despite customization, we are aware there might be scenarios where automatically-produced results might not be enough nor desirable. In [Cervantes et al. 2017], the authors conclude that fully automated methods might lead to overgeneralized solutions that are not always adequate. To support those situations where automatic repairing is not enough, we integrate a module through which users can provide PARMOREL with their feedback of the found solutions. This way, we keep users in the loop while PARMOREL learns, so user feedback can shape future repair solutions.

To show the extensible potential of PARMOREL, in this paper we focus on customizing the repair so that the produced models are as close as possible to the original model structure since, as above-mentioned, this is an important problem highlighted in the literature. We achieve this with the addition of a model distance component to the PARMOREL architecture. This component implements a model comparison mechanism [Stephan and Cordy 2013] in order to compute the distance between two models. This integration leads to the production of models that are repaired respecting the original model structure as much as possible. By using model distance, the approach takes model repair one step further in minimizing the undesired choices and side-effects that automatic approaches produce.

We validate the integration of this component with a dataset of 107 models crawled from Github repositories containing Ecore models. The results are encouraging, including additional preferences to the framework can lead to better precision in selecting the best repaired models in terms of the specified preferences. In our experiment we tested PARMOREL with two preferences, i.e., model quality + model distance, that have been both integrated with model-based artifacts in PARMOREL.

Structure of the paper. This paper is organised as follows: Section 2 shows real examples of corrupted models taken from Github repositories. Section 3 presents our approach. Section 4 demonstrates how PARMOREL can integrate different customizable preferences in order to perform the repair process with better precision. As an example, in Section 5 we extend the preferences in PARMOREL by using a model distance metric. In Section 6 we evaluate if PARMOREL can produce higher results when extending its components, following the example of the previous section. Then, we present threats to validity in Section 7, explore the related work in Section 8 and conclude the paper in Section 9.

2 REPAIRING MODELS

As any other software artifacts, domain models can be subject to modifications, to address changes in software requirements, to improve model quality characteristics [Basciani et al. 2016], or changes on the size of the conceptual domain to be engineered. During these modifications the chances of corrupting a model increases, specially with collaborative activities [Di Rocco et al. 2015; Franzago et al. 2017]. To better explain the problem, we are facing in this paper,

Table 1: Occurrences of errors in the selected dataset

	Error	Occurrences
E1	The opposite of a transient reference must be transient if it is proxy resolving	2
E2	The opposite must be a feature of the reference's type	1
E3	The opposite of the opposite of a reference must be the reference itself	5
E4	Not transient Attribute so it must have a data type that is serializable	7
E5	A primitive type cannot be used in this context	4
E6	Two or more Classifier with the same name	2
E7	Two or more feature with the same name	20
E8	Invalid specified literal	166
E9	Not well formed name	216
E10	Operation with the same signature as an accessor method	5
E11	A containment or bidirectional reference must be unique if its upper bound is different from 1	160
E12	The same contained instance cannot be contained in two different instances	94

we analyzed models retrieved from Github repositories. To retrieve these models we rely on the dataset used in [Nguyen et al. 2019] and filtered in order to get only corrupted models. From this selection we have worked with 107 models, where errors were distributed as in Table 1. We identified 12 errors, E1–E12, supported by our tool PARMOREL, in the following we detail three of these errors

since we will focus on them in a later section. Table 1, with a total of 973 error occurrences confirms the need to support model repair as an automated activity. Each of these errors can be repaired by several actions. Hence, this also confirms the need to support modelers with extensible repair preferences, so that they can customize which kind of repair actions satisfy their specific needs.

In the following we report an excerpt of the models affected by 3 of the 12 errors, and we briefly describe each error and the repairing actions that could solve it. These 3 errors will be part of examples later in the paper and hence, we focus on them now as a representative sample. The complete list and explanation of the errors and models where occurred can be found at https://github.com/models2020modelsrepair/ModelsRepair.git

E6. Two or more classifiers with the same name cause error E6, present twice in the dataset. Specifically, the model car.ecore contains two classes with same name but different letter casing. Precisely, AirCond and Aircond inheriting from the same classifier can be seen in Fig. 1. This leads to the hypothesis that the modeler did not check the model and added the same class twice during the development process. The multiple resolutions for this error include renaming or removing any of the classifiers.

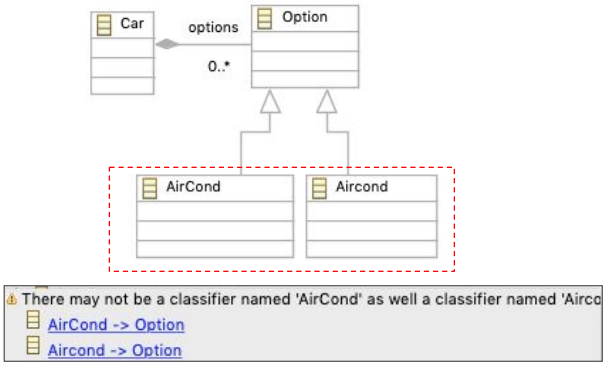


Figure 1: E6 identified in car.ecore model

E8. Error E8 "Invalid specified literal" is quite widespread in the analyzed dataset, 166 times. This error basically is a warning saying that the default value specified is not coherent with the literals specified in the enumeration.

Indeed in this case the default value specified for the attribute limitType is 1, when the literals on the enumeration, set as datatype, are the following: Reporting, Hard, SoftLinear, SoftQuadratic. Maybe the developer wanted to specify Hard as default value, being at the position 1 of the possible literals. Possible solutions are to modify the literal value in the attribute with any of the possible literals, modify the default in the datatype enumeration, etc.

E11. This error is one of the most widespread and it is about setting a unique containment reference if the upper bound is different from 1. In this model reported in Fig. 3, the containment reference xMLNSPrefixMap and xSISchemaLocation are not declared as unique, but their upperBound is set to -1, violating this

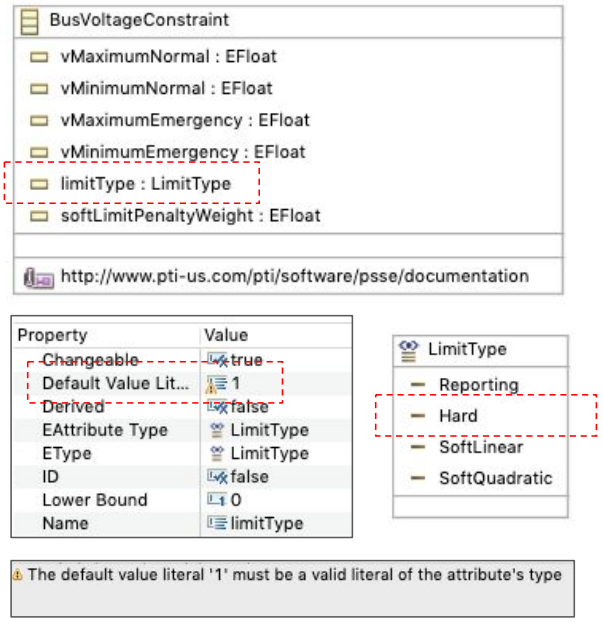


Figure 2: E8 identified in OPF31.ecore model

uniqueness constraint of a containment relationship. For this reason, the possible resolutions are various, e.g., set the upperbound to 1, set the unique property, delete the faulty reference or unset the containment. This error is quite widespread since 160 occurrences have been matched in the dataset.

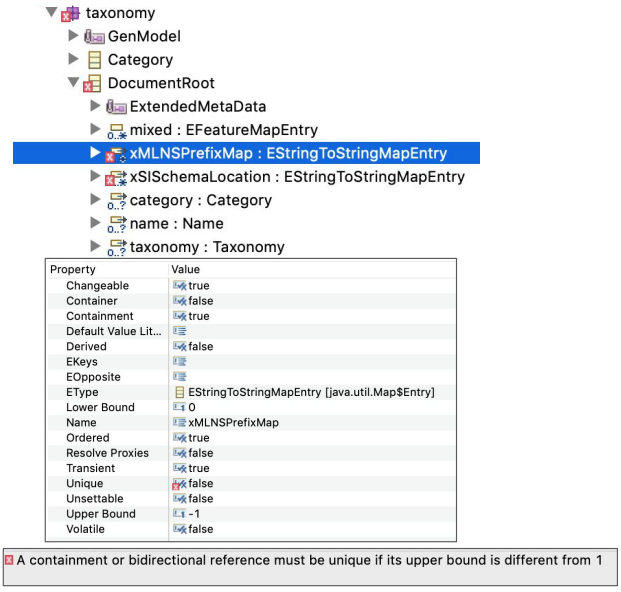


Figure 3: E11 identified in taxonomy.ecore model

Since all these errors can lead to multiple repair actions of the corrupted models, we propose a mechanism to select the best action for each supported repairing mechanism based on preferences customized by the users. This mechanism is extensible and embedded in PARMOREL as mentioned before and will be explained in the next sections.

3 APPROACH

PARMOREL uses RL algorithms to find which is the best possible repairing action for each error in the model given as input. RL consists of algorithms able to learn by themselves how to interact in an environment without existing pre-labelled data, only needing a set of available actions and rewards for each of these actions. We report the abstract architecture in Fig. 4 with the main macro components and artifacts involved in the process of model repair. PARMOREL internally relies on a modeling framework to detect issues in the corrupted model (e.g., see Section 2) given as input. If the validation check performed by the modeling framework is not successful, the repairing process starts. The embedded modeling framework is also responsible for applying the repairing actions selected by PARMOREL and creating the repaired model returned as output of the entire process. The learning algorithm allows PARMOREL to repair without having any prior data about repairing models (labelled data, historical data, etc). By using and tuning RL rewards, these algorithms can learn which are the best actions to repair a given error. RL rewards can be adapted to align with any preference introduced by the user as long as it can be quantified (e.g., preserve the original model structure by minimizing the model distance metric or boost quality characteristics by maximizing quality metrics). Preferences need to be quantified so that their values can be directly mapped into RL rewards. For example, the value of the model distance itself could be used as a reward. These preferences can be considered singularly, or in combination, and the proposed architecture is open to support additional reward mechanisms.

Before finding a repair sequence for a given model, PARMOREL is executed for a number of episodes. Each episode equals one iteration attempting to repair the model in where different actions will be applied to repair the different errors present in the model. For each of these episodes, a possible repair sequence is found, and applying it, a provisional repaired model is created. The provisional repaired models are analyzed according to the preferences selected and the result is translated into rewards (e.g., depending on how close the provisional model is to the original one it will get a higher or lower reward). Hence, PARMOREL can identify how good each applied repairing action is according to the user requirements. Following this process, after each episode, actions leading to the results closest to the user requirements will have higher rewards and thus more probabilities of being selected.

Additionally, for those situations where automatic repair might not be enough for the users, they can manually select which sequence of actions they prefer among the repair sequences found in the episodes. By doing this, the algorithm provides extra rewards to the selected actions. This way, users can correct and influence how the algorithm learns. Strong interaction with the user requires more effort from her and also higher computational effort [Cervantes et al. 2017] hence, the interaction we provide is optional

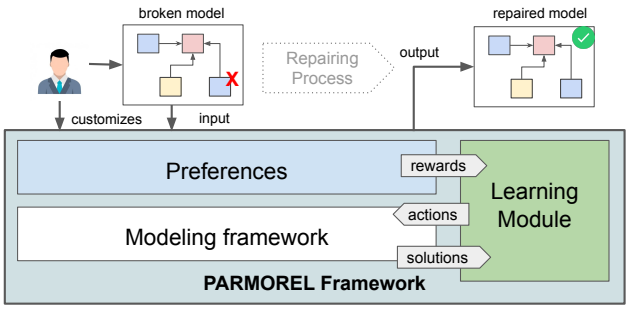


Figure 4: PARMOREL Components

and it takes place only after all episodes have ended. Furthermore, by providing feedback at the end of the execution, users can compare the provisional repaired models and the sequences of actions that produced them. So their choice is done after measuring the consequences of choosing either. For example, PARMOREL finds two sequences of actions seq1 and seq2. Although seq2 produces a model slightly closer to the original, after checking the results of seq1 the user sees that it produces a result that he prefers over minimizing the distance. The actions in seq1 will then be rewarded and PARMOREL will prioritize them in future repairs.

After performing enough repairing iterations, PARMOREL will select the repair sequence with higher rewards (with or without user feedback) and saves the final repaired model.

4 PARMOREL EXTENSIBLE FRAMEWORK

Our approach is based on three main components: a modeling framework, a learning module and user preferences, see Fig 4. PARMOREL is designed as an Eclipse plugin, hence, is extensible and users can customize these three components through a series of interfaces. Now, we will go through each of these components.

Modeling framework: The modeling framework validates the models and provides PARMOREL with the errors they present and actions available for editing them. In the examples in this paper, the modeling framework integrated in PARMOREL is the Eclipse Modeling Framework (EMF) [Steinberg et al. 2008]. However, in PARMOREL, the concepts of model, actions and errors are abstract, and they can be implemented by different modeling frameworks without affecting other parts of the system. Hence, working with other type of models would also be possible, by connecting frameworks such as Kermeta [Jézéquel et al. 2009] or MetaEdit+ [Tolvanen and Kelly 2009].

Learning module: The learning module is responsible for learning which actions are the best to repair the errors in the models according to the preferences introduced by the users. RL is a broad field with many algorithms, currently, PARMOREL works with Q-learning [Thrun and Littman 2000]. Q-learning provides several features that are useful to solve the model repair problem in terms of reusability, structure and decision making. In Q-learning, knowledge acquired is stored in a table structure called Q-Table. This table stores pairs of states and actions together with a Q-value. The Q-value is calculated using the rewards and it indicates how good each pair is. The Q-value is obtained with repeated calculations

based on the Bellman Equation [Bellman 2013] (see Equation 1), telling that the maximum future reward is the reward r the agent received for entering the current state s_t with some action a_t plus the maximum future reward for the next state s_{t+1} and action a_{t+1} reduced by a discount factor γ. This allows inferring the value of the current (s_t, a_t) pair based on the estimation of the next one s_{t+1}, which can be used to calculate an optimal policy to select actions. The factor α provides the learning rate, which determines how much new experience affects the Q-values. One of the variables used to calculate the Q-value, is the maximum weight stored in the Q-table for the next error to repair ($\max_{a'} Q(s_{t+1}, a')$). This allows us to measure the consequences of applying a certain action in the model (e.g., if applying an action creates a new unknown error this action will be punished, getting a lower weight). At the end of the execution, pairs with the highest Q-value will conform to the policy to solve the problem. Our algorithm is epsilon-greedy (ε-greedy): it avoids local optima using an exploration-exploitation tradeoff by exploring (i.e. choosing a random action) with probability ε, and exploiting (i.e. choosing the action with highest Q-value) the remainder of the time. We work with an ε of 0.3. Regarding other parameters, discount factor (γ), and learning rate (α), we use 1.0 for both of them. According to our experiments the best results were obtained with these values.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s,a)) \quad (1)$$

However, the algorithm can be changed by another algorithm, as long as it corresponds with a finite Markov Decision Process (MDP) [Thrun and Littman 2000] and supports the concepts of states (errors in our problem), actions and rewards. This way, Q-Learning could be substituted with other RL algorithms.

Preferences: Users can customize the results PARMOREL produces with their own preferences. PARMOREL supports preferences as long as they can be translated into numeric values. PARMOREL will take these values as rewards that will guide the repair process. For example, users could prefer to repair priorizing a quality characteristic (maintainability, reusability, understandability, etc), to minimize the model distance with respect to the original model, to minimize the impact of bad smells in the model, etc. This extension can be done by linking tools such as [Addazi et al. 2016] for model distance, [Bettini et al. 2019] or for quality characteristics [Fourati et al. 2011]. Also the relation with other modeling artifacts could be considered as existing transformations [De Lara et al. 2017] defined on top of the corrupted models. PARMOREL will use the rewards to estimate how good or bad each action is to satisfy the user preferences.

With this extensible approach, PARMOREL can be adapted to the needs of the users in terms of different models, preferences and learning algorithms, so it can be adapted to new problems and situations.

One of the advantages of using RL is that these algorithms improve their performance the more they are applied. In our approach, the more models are repaired the better the performance becomes since PARMOREL acquires and builds experience that is reused in later repairs. To this extent, and to support the extensibility potential of our approach, we define experience as a model (see Fig. 6)

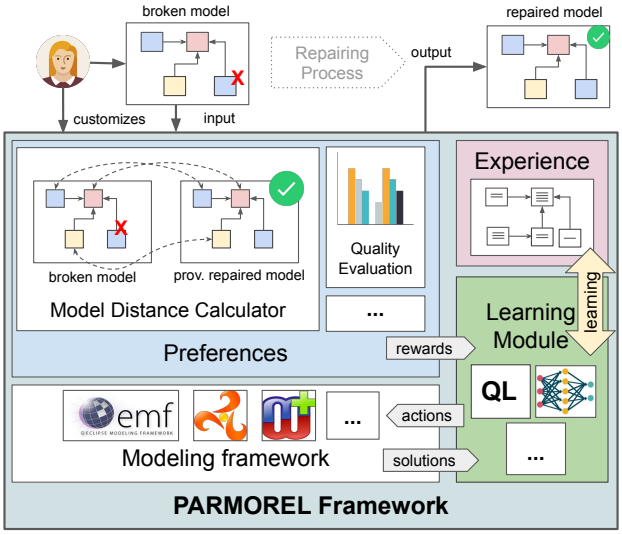


Figure 5: PARMOREL Extensible Components

that can be reused regardless of the modeling framework, learning module and preferences selected by the user.

Experience: The learning information gained after each repair is represented by the concept Experience which is composed of one to many entries and preferences. Reward and Preference are linked so that PARMOREL can reuse the rewards corresponding with a specific preference the next time it is selected by the user. The concept of Entry includes all the information PARMOREL learns, this is: which Action can repair an Issue in a given Location in the model and how good is that action in terms of a Reward. In this model we refer to *issues* instead of *errors* since PARMOREL could be configured to fix bad smells or inefficient patterns, which are not errors.

By implementing the concepts in this model, the framework becomes adaptable to different repair scenarios. For example, if a user repairs a series of models based on EMF, the acquired experience should be reusable when repairing conceptual models regardless the framework. For instance, conceptually, a class diagram can act as a metamodel for an object diagram. For this reason we could reuse the accumulated experience to repair corrupted class diagrams.

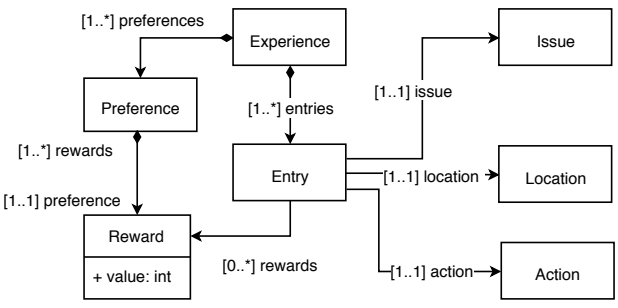


Figure 6: Model of experience in PARMOREL

Additionally, by combining this model with the machine learning technique of transfer learning [Barriga et al. 2020], what is learnt from the repair of one user could be reused by other users. With this, consequent executions of PARMOREL, even by different users, could achieve better performance the more experience is reused. For more details about how PARMOREL uses transfer learning we refer the reader to our previous work [Barriga et al. 2020].

5 EXTENDING PARMOREL WITH MODEL DISTANCE

In this section, we exemplify an extension of the user preferences by using a model distance metric to repair the models. By using this metric we can reward the preservation of the original model structure when repairing, minimizing undesired side-effects in the repaired model.

Counting model differences is a challenging problem in MDE, especially when large sets of models have to be compared. The task of comparing two or more models can be managed by specific distance metrics, inspired by distances between words and graphs [Ferdjoukh et al. 2017]. In this work we use a distance metric in order to understand how much a repaired model is close to the initial broken model. We have implemented this mechanism in the component Model Distance Calculator, shown in Fig. 5, that basically compares the two models, the broken one with the provisional repaired one produced in each episode and gives a distance reward to the Experience module. The Model Distance Calculator module is implemented as an Eclipse plugin, composed of a model matching algorithm specified with an ECL script, reported in Listing 1, and it is invoked for all the model pairs in order to calculate their distance metric. Table 2 represents an example calculation where in the *Basic* column we reported the implementation shown in Listing 1, we will define *basic*, for the distance calculation.

Table 2: Distance Matrix for taxonomy.ecore example and error E11

		Distance	
Repaired model	**Actions**	**Basic**	**Custom**
model 1-1	E11-1: unset containment E11-2: unset containment	1.0	0.89
model 1-2	E11-1: upperBound changed E11-2: upperBound changed	0.89	0.89
model 1-3	E11-1: upperBound changed E11-2: removed reference	0.89	0.89
model 1-4	E11-1: set uniqueness E11-2: set uniqueness	1.0	0.89

This distance metric is generated for a corrupted model with each of its corresponding repaired versions, where the value indicates how much each pair is similar. Distance value goes from 0 to 1.0 with 1.0 meaning that models are structurally the same; i.e., the closest distance possible. For instance, the model org.eclipse.wst.ws.internal-model.v10.taxonomy.ecore (model 1 for sake of shortness) from our dataset contains multiple errors, including E11 (see Fig. 3) twice. As explained in Section 2, for this error PARMOREL finds 4 possible solutions: setting the upperbound to 1, setting the unique property, deleting the faulty reference or unsetting the containment. By combining these actions for the two occurrences of E11, we obtain 16 different repair sequences. When comparing the models produced by these sequences with the original model, we obtain 2 different distance metric values. Hence, in this section we focus on 4 of the provisional repaired models produced (model 1-1, 2, 3 and 4).

In the example in Table 2 (Basic column), model 1-1 and model 1-4 are the solutions with closest distance, since their actions modify the uniqueness property of the faulty reference and the containment attribute, and those properties are not considered in the comparison matching (see lines 33-34 of Listing 1). Oppositely, model 1-2 and model 1-3 presents the furthest distance, since PARMOREL applies two actions that modify components considered in the matching comparison: a modified upperbound and the removal of a reference.

```
1 pre variables {
2     var simmetrics : new Native('org.epsilon.ecl.tools.
          ↪ textcomparison.simmetrics.SimMetricsTool');
3 }
4 rule EClass
5   match s : Source!EClass
6   with v : Target!EClass {
7
8   compare {
9     if(s.name.fuzzyMatch(v.name)){
10      return true;
11    }else{
12      return false;
13    }
14  }
15 }
16 rule EAttribute
17   match s : Source!EAttribute
18   with v : Target!EAttribute {
19
20   compare {
21    if(s.name.fuzzyMatch(v.name) and s.etype.isDefined() and v.
          ↪ etype.isDefined() and s.etype.name.fuzzyMatch(v.
          ↪ etype.name) and s.eContainingClass.name.fuzzyMatch(
          ↪ v.eContainingClass.name)){
22      return true;
23    }else{
24      return false;
25    }
26  }
27 }
28 rule EReference
29   match s : Source!EReference
30   with v : Target!EReference {
31
32   compare {
33    if(s.name.fuzzyMatch(v.name) and s.etype.name.fuzzyMatch(v.
          ↪ etype.name) and s.eContainingClass.name.fuzzyMatch(
          ↪ v.eContainingClass.name) and s.lowerBound==v.
          ↪ lowerBound and s.upperBound==v.upperBound){
34      return true;
```

```
35     }else{
36       return false;
37     }
38   }
39 }
40 ...
```

Listing 1: Fragment of the ECL implementation of the basic matching algorithm

As we said, the implementation of the model distance calculator consists of two main components, 1) the matching algorithm implemented in ECL and reported in part in Listing 1, and 2) the calculation of the distance value depending on the matched elements. We do not report this second component since it is implemented in Java and can be easily imagined. This basic matching algorithm was implemented for general purposes, and it uses the Levenshtein[1] edit distance [Levenshtein 1966] when calculating the name similarity of different elements such as classes (lines 4–15) and structural features (lines 16–39) of the model. This ECL script can be customized in order to add other constraints or relax the similarity function, e.g., removing the lower bound and upper bound matching for the structural features (for instance line 21). This customization will affect the distance calculation and in turn affects the repairing sequences chosen by PARMOREL. Indeed, by further restricting the comparison mechanism as we propose in Listing 2, the comparison algorithm will match elements differently, returning the distance results in Table 2 in column *Custom*. This customization clearly shows how to customize the comparison mechanism for the matching strategies for the structural features and can offer an additional way to the user to implement her own preference.

```
1 ...
2 rule EReference
3   match s : Source!EReference
4   with v : Target!EReference {
5   compare {
6    if(s.name.fuzzyMatch(v.name) and s.etype.name.fuzzyMatch(v.
         ↪ etype.name) and s.eContainingClass.name.fuzzyMatch(
         ↪ v.eContainingClass.name) and s.lowerBound==v.
         ↪ lowerBound and s.upperBound==v.upperBound  and s.
         ↪ unique==v.unique and s.containment==v.containment){
7      return true;
8    }else{
9      return false;
10   }
11   }
12 }
13 ...
```

Listing 2: Customized comparison strategy for references of the matching algorithm

The outcome of such a phase is a matching model given as input to the Java method that simply builds the distance values. Just to give the intuition of how this value is computed we report in (2) the final formula that will give the output distance value of the two given models[2]:

$$distance = ((((classsim)/nrclasses)+ \\ ((featuresim)/nrfeats))/2); \quad (2)$$

Basically, the *distance* is computed as the sum of the class similarity on the total number of classes, features similarity on the number of features. PARMOREL obtains the distance metric from the Model Distance Calculator, and uses its value as a reward. This way the framework will learn to repair in a way that produces models as close as possible to the original.

If we explore the example reported in Table 2, multiple resolutions are possible to fix the corrupted model but PARMOREL will pick by default the first one with closest distance, i.e., model 1-1 in both cases of the basic and the custom algorithm. With the custom comparison mechanism, the distance values changed since in Listing 2 the comparison mechanism at line 6 considers also the *unique* and the *containment* properties that are used in the applied repairing strategies of model 1-1 and 1-4. Figure 7 shows an example of a repaired model for E11 (see Fig. 3), where the corruption points are then resolved. It is worth noting how a tiny modification

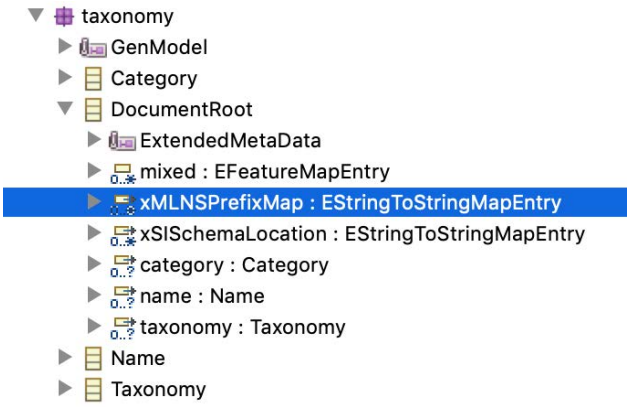


Figure 7: Repaired example model 1-1 of taxonomy.ecore

to the distance calculation script can affect the reward mechanism and so the selected repaired models by the algorithm. Adding additional preferences to PARMOREL can further improve the repairing mechanism.

6 EVALUATION

In this section, we present an evaluation of the proposed approach focusing on extensibility. We focus on evaluating if, by extending the preferences, the framework is able to improve the precision in selecting better repaired models. First, we introduce how the considered quality characteristics are linked to model elements [Genero and Piattini 2001]. For instance, the maintainability of a model is influenced by the size of the model and then the number of classes;

[1] This functionality is loaded at line 2 as a native Java library

[2] The interested reader can refer to [Di Ruscio et al. 2020] for a complete implementation of the similarity function that we borrowed for our implementation

understandability is influenced by number of hierarchies, etc. For this reason, if we consider an error impacting specific model elements, PARMOREL should produce a repaired model optimizing the quality characteristics that are influenced by the repaired elements. For example, if we consider error E6, where we have two classes with same name in the model, if one of the classes is also involved in a hierarchy, fixing this error could impact all the quality characteristics considering the number of hierarchies in the model. As a consequence, if we compare the basic implementation of the model distance (Listing 1) with the customized one (Listing 2), even if the customization of the matching strategy is minimal, the selected repaired model should have improved quality characteristics since the distance calculation is more refined.

Hence, we proceed to run PARMOREL first with the basic implementation of the distance calculation and then the customized one; if our hypothesis is correct we should have better precision in selecting the repaired model and consequently optimize quality characteristics.

As an example, we focus on improving the complexity of the models. For this, we will repair models from the dataset containing error E11. This error is related to containment references and the upper bound and uniqueness of the reference—all these affect complexity. Table 3 shows the models in our dataset impacted by error E11. *Complexity* is defined in terms of the number of static relationships between the classes (i.e., number of references). The complexity of the association and aggregation relationships is counted as the number of direct connections, whereas the generalization relationship is counted as the number of all the ancestor and descendant classes. Thus, the complexity quality characteristic can be defined as follows:

$$Complexity = (NR - NUR + NOPR + UND + (NR - NCR)) \quad (3)$$

where NR is the total number of references, NUR is the number of unidirectional references calculated as the difference between bidirectional and total reference number, NOPR is total number of opposite references, NCR is the total number of containment references, and UND is the understandability value calculated as defined in equation 6 (see below). According to the given definition, the lower the value for the complexity characteristic the better.

Table 3 reports the complexity value after repairing with the basic distance and the customized one. For all the cases, the complexity improved (✓) (decreased, so it is optimized) or at least remained unchanged (=). The results are that 12 of the selected repaired models improved the complexity and 7 remained unchanged, confirming our hypothesis.

Considering the customized distance algorithm extension, the quality characteristics that are improved in relation to the whole dataset are reported in Table 4[3]. Maintainability has remained unchanged or improved in the 80.2% of the total models in the dataset, reusability in 84%, complexity in 84% and understandability in 100%[4]. The unimproved cases depend on the occurring errors in the models and on the model elements that affect the quality characteristics. For this reason, we also report that the most widespread error in the unimproved models is E8, in order to discuss why the quality has not been improved by selecting the best repaired model in terms of distance by using the basic and the custom distance algorithms.

[3]The complete results are available as a Google spreadsheet and the dataset of models can be found at https://Github.com/models2020modelsrepair/ModelsRepair

[4]Some of the cases are excluded since the quality evaluation exited with errors or warnings

Table 3: Repairing models with error E11 while optimizing their complexity with the custom distance calculator

	complexity		
model	**basic**	**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	3.3	1.33	✓
pom.ecore	19.03	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	=

Recall that the distance function has only been customized for the references' matching. As a consequence, only quality characteristics which are calculated using references are the ones that are impacted. In fact, we can verify that for error E8 and the maintainability quality characteristics—where all the components of formula 3 are number of classes, structural features, hierarchies and reference siblings—the custom distance calculation did not optimize or affect this maintainability.

Error E8 is the most widespread error in cases where quality characteristics are unchanged, and it represents an invalid specified literal in the model, which means that fixing the error does not affect the maintainability, since enumerations are not considered in the formula.

$$Maintainability = \left(\frac{NC + NA + NR + DIT_{Max} + Fanout_{Max}}{5}\right) \quad (4)$$

Regarding reusability, it has improved in 84% of the cases, and is defined with the following formula:

$$Reusability = AIF = \left(\frac{INHF}{NTF}\right) \quad (5)$$

where *INHF* is the sum of the inherited features in all classes, and *NTF* is the total number of available features. E8 is again the most present error in unimproved cases, and it does not affect the formula, except when the attribute where the error is matched is the one with the invalid specification.

The same reasoning applies to complexity in the sense that in the unimproved cases, it is because the error is matched on features which are not reflected in the quality calculation.

Table 4: Percentage of models which, after repairing with closest distance preference, are improved with respect to quality characteristics

Quality Attribute	Improved
Maintainability	80.2%
Reusability	84%
Complexity	84%
Understandability	100%

Finally, understandability is improved in all the models (100%), being formulated as:

$$Understandability = \left(\frac{\sum_{k=1}^{NC} PRED + 1}{NC} \right) \quad (6)$$

where PRED is the number of predecessors. For most models this quality attribute remains stable, this is caused because fixing errors in this dataset does not affect hierarchies and hence PRED remains unchanged. We can confirm that a quality characteristic improves only in cases where the model repair impacts elements which are used in the quality characteristics calculation.

With the results of this evaluation, we can conclude that, by extending PARMOREL preferences, the precision of the framework improves and it is able to produce repaired models with higher quality characteristics.

7 THREATS TO VALIDITY

Internal threats. Among the internal threats we have the quality evaluation process since the quality model is user-defined. Quality aspects are often based on the modeler's experience and mistakes in these quality models' definitions may impact the results. Including experts in the quality definition process can mitigate this aspect, e.g., by including definitions adopted from the literature. In this direction, the considered formulas in the previous section are based on the literature. Our distance metric calculation is parametric with respect to a match threshold, specified in the *FuzzyMatch* function, that in our case is set to 0.5. Varying this parameter, the distance calculator may return different results, so we set this parameter to a value that in our experiments seems to be balanced enough in returning accurate results. We plan to extend our experiments with sliding this value in order to offer the users of PARMOREL a way to set her own preference also in this case.

External threats. The dataset we used for the experiment can be considered as a potential external threat to the validity of the evaluation. We have used a dataset of 107 corrupted models. The number of models is not large for the standard evaluation but finding real corrupted models on existing repositories is not an easy task. However, this threat is justified by the heterogeneity of the sources and the authors of the models, that are distributed; in fact these models have been obtained from different Github repositories.

Also, throughout the paper we have picked four quality characteristics (maintainability, understandability, reusability and complexity) as a proof of concept to measure the quality of the repaired models when extending PARMOREL. Although many other characteristics can be measured, we consider this set representative enough since they are related to different elements in the models.

Finally, the examples in the paper are based on EMF and Ecore models, but as we explained, it is possible to switch to other modeling frameworks by extending PARMOREL. Within EMF, the work presented in this paper is is specific for Ecore models. However, it could be applied in general to models instances if the repairing actions retrieved from the framework were domain specific.

8 RELATED WORK

The main features that distinguish our approach from other model repair approaches is the extensibility of the framework and the capability to learn from each repaired model in order to streamline the performance. We could not find in the literature any research applying RL to model repair nor providing our degree of customization. The most similar work to ours we could find is [Puissant et al. 2015], where Puissant et al. present Badger, a tool based on an artificial intelligence technique called automated planning. Badger generates plans that lead from an initial state to a defined goal, each plan being a possible way to repair one error. We prefer to generate sequences to repair the whole model, since some repair actions can modify the model drastically, and we consider it counter-intuitive to decide which action to apply without knowing its overall consequences, additionally, RL performs better after each execution.

Nassar et al. [Nassar et al. 2017] propose a rule-based prototype where EMF models are automatically completed, with user intervention in the process. Our approach allows for more autonomy since preferences are only introduced at the beginning of the repair process and user feedback at the end of all episodes, requiring less effort from the user.

In this direction, authors in [Cervantes et al. 2017] present an interactive repairing tool powered by visual comparison of models performing conformance checking. They conclude that fully automated methods lead to overgeneralized solutions that are not always adequate, and strong interaction comes with a high computational effort, therefore as future work they seek an equilibrium between automation and interaction. This is our vision: balance between the algorithm independence and user intervention to provide personalized solutions.

Taentzer et al. [Taentzer et al. 2017] present a prototype based on graph transformation theory for change-preserving model repair. The authors check operations performed on a model to identify which ones caused inconsistencies and apply the correspondent consistency-preserving operations, maintaining already performed changes on the model. Their preservation approach is interesting, however it only works assuming that the latest change of the model is the most significant.

Kretschmer et al. introduce in [Kretschmer et al. 2018] an approach for discovering and validating values for repairing inconsistencies automatically. Values are found by using a validation tree to reduce the state space size. Trees tend to lead to the same solutions once and again due to their exploitation nature (probing a limited region of the search space). Differently, RL algorithms include both exploitation and exploration (randomly exploring a much larger portion of the search space with the hope of finding

other promising solutions that would not be selected normally), allowing to find new and, sometimes better optimized fixes for a given problem.

It is worth mentioning search-based and genetic algorithm-based approaches since, although they have not been applied yet to model repair, they are possible competitors to RL. These techniques have showed promising results dealing with model transformations and evolution scenarios, for example in [Kessentini et al. 2017] authors use a search-based algorithm for model change detection. These algorithms deal efficiently with large state spaces, however they cannot learn from previous tasks nor improve their performance. While RL is, at the beginning, less efficient in large state spaces, it can compensate with its learning capability. At the beginning performance might be poor, but with time repairing becomes straightforward.

Lastly, another search-based approach is presented by Moghadam et al. in [Moghadam and Ó Cinnéide 2011]. In this work, authors present Code-Imp, a tool for refactoring Java programs based on quality metrics that achieves promising results at code-level by using hill-climbing algorithms [Selman and Gomes 2006]. These algorithms are interesting to find a local optimum solution but they do not assure to find the best possible solution in the search space (the global optimum). By using RL we assure to find the global optimum aligned with the user preferences, in our example the sequence of repairing actions that minimizes the distance with respect to the original model.

9 CONCLUSIONS AND FUTURE WORK

In this paper we presented PARMOREL, an extensible framework for model repair based on three main components: a modeling framework, a learning module and user preferences. Users can customize the modeling framework to work with different types of models, the preferences to obtain different customized repairs and the learning module to use different learning algorithms. Supported algorithms must implement a Markov Decision Process [Thrun and Littman 2000] and support the concepts of states (errors in our problem), actions and rewards.

As an example, we have extended the preferences using a model distance metric. To evaluate if the extensibility of the framework had any impact in the repaired models' quality, we applied two versions of the distance metric (a basic and a customized one) to repair models in a dataset extracted from Github. Our hypothesis was that, by extending PARMOREL, we could achieve models with better quality. Our results concluded that using the customized distance metric PARMOREL was able to produce models of higher quality. We measured the complexity, understandability, maintainability and reusability of the repaired models, obtaining better (or stable) results in the majority of them.

Next, we plan to provide further testing on extending the modeling framework. We will test the framework with a more extended dataset of domain models and errors, with the help of modelers that may attest if the repaired sequence really offers better quality of the repaired domain model. In particular we plan to test the presented approach with a bigger dataset of domain models coming from third party repositories in order to validate the approach with more real-world examples. Additionally, we plan to create a benchmark with the mentioned dataset, with which we will compare PARMOREL results and its performance to other existing model repair approaches.

Also, we will perform a comparative testing on how PARMOREL performs using different learning algorithms within RL. We have a special interest in comparing the tool performance with other automatic repairing tools presented in Section 8, paying especial attention to search-based, rule-based and automated planning approaches.

We plan to perform a study to assess to which extent experience can be reused when changing the framework and the type of models, for example, reusing the accumulated experience of repairing structural models to repair corrupted behavioral models.

REFERENCES

Lorenzo Addazi, Antonio Cicchetti, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2016. Semantic-based Model Matching with EMFCompare.. In *ME@ MODELS*. 40–49.

Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2019. Personalized and Automatic Model Repairing using Reinforcement Learning. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*. 175–181. https://doi.org/10.1109/MODELS-C.2019.00030

Angela Barriga, Adrian Rutle, and Heldal Rogardt. 2020. Improving model repair through experience sharing. *Journal of Object Technology* 19, 1 (2020).

Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2016. A customizable approach for the automated quality assessment of modelling artifacts. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 88–93.

Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2019. A tool-supported approach for assessing the quality of modeling artifacts. *Journal of Computer Languages* 51 (2019), 173–192.

Richard Bellman. 2013. *Dynamic programming*. Courier Corporation.

Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2019. Quality-Driven detection and resolution of metamodel smells. *IEEE Access* 7 (2019), 16364–16376.

Barry W Boehm, John R Brown, and Mlity Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 592–605.

Abel Armas Cervantes, Nick RTP van Beest, Marcello La Rosa, Marlon Dumas, and Luciano García-Bañuelos. 2017. Interactive and incremental business process model repair. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 53–74.

Juan De Lara, Juri Di Rocco, Davide Di Ruscio, Esther Guerra, Ludovico Iovino, Alfonso Pierantonio, and Jesús Sánchez Cuadrado. 2017. Reusing Model Transformations Through Typing Requirements Models. 264–282. https://doi.org/10.1007/978-3-662-54494-5_15

Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2015. Collaborative Repositories in Model-Driven Engineering. *IEEE Software* 32, 3 (2015), 28–34. https://doi.org/10.1109/MS.2015.61

Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio, and Lorenzo Bettini. 2020. Detecting metamodel evolutions in repositories of MDE projects. In *Modelling Foundations and Applications*. Springer International Publishing. to appear.

R. Geoff Dromey. 1995. A model for software product quality. *IEEE Transactions on software engineering* 21, 2 (1995), 146–162.

Adel Ferdjoukh, Florian Galinier, Eric Bourreau, Annie Chateau, and Clémentine Nebut. 2017. Measuring differences to compare sets of models and improve diversity in MDE. In *ICSEA: International Conference on Software Engineering Advances*.

Rahma Fourati, Nadia Bouassida, and Hanêne Ben Abdallah. 2011. A metric-based approach for anti-pattern detection in UML designs. In *Computer and Information Science 2011*. Springer, 17–33.

Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. 2017. Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering* 44, 12 (2017), 1146–1175.

Marcela Genero and Mario Piattini. 2001. 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*.

Ludovico Iovino, Angela Barriga, Adrian Rutle, and Heldal Rogardt. 2020. Model Repair with Quality-Based Reinforcement Learning. *Journal of Object Technology* 19, 2 (July 2020), 17:1–21. https://doi.org/10.5381/jot.2020.19.2.a17

Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. 2009. Model driven language engineering with kermeta. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 201–221.

Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. 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, 163–172.

Marouane Kessentini, Usman Mansoor, Manuel Wimmer, Ali Ouni, and Kalyanmoy Deb. 2017. Search-based detection of model level changes. *Empirical Software Engineering* 22, 2 (2017), 670–715.

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

Roland Kretschmer, Djamel Eddine Khelladi, and Alexander Egyed. 2018. An automated and instant discovery of concrete repairs for model inconsistencies. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 298–299.

VI Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.

Jesús J López-Fernández, Esther Guerra, and Juan De Lara. 2014. Assessing the Quality of Meta-models.. In *MoDeVVa@ MoDELS*. Citeseer, 3–12.

Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. 2013. Model repair and transformation with Echo. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 694–697.

Iman Hemati Moghadam and Mel Ó Cinnéide. 2011. Code-Imp: a tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*. 41–44.

Nebras Nassar, Hendrik Radke, and Thorsten Arendt. 2017. Rule-Based Repair of EMF Models: An Automated Interactive Approach. In *International Conference on Theory and Practice of Model Transformations*. Springer, 171–181.

Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Alfonso Pierantonio, and Ludovico Iovino. 2019. 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, 272–282.

Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2018. ReVision: a tool for history-based model repair recommendations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 105–108.

Maryoly Ortega, María Pérez, and Teresita Rojas. 2003. Construction of a systemic quality model for evaluating a software product. *Software Quality Journal* 11, 3 (2003), 219–242.

Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. 2015. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling* 14, 1 (2015), 461–481.

Bart Selman and Carla P Gomes. 2006. Hill-climbing Search. *Encyclopedia of cognitive science* (2006).

Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

Matthew Stephan and James R Cordy. 2013. A Survey of Model Comparison Approaches and Applications.. In *Modelsward*. 265–277.

Misha Strittmatter, Georg Hinkel, Michael Langhammer, Reiner Jung, and Robert Heinrich. 2016. Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. (2016).

Eugene Syriani, Robert Bill, and Manuel Wimmer. 2019. Domain-Specific Model Distance Measures. *Journal of Object Technology* 18, 3 (2019).

Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. 2017. Change-preserving model repair. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 283–299.

Sebastian Thrun and Michael L Littman. 2000. Reinforcement learning: an introduction. *AI Magazine* 21, 1 (2000), 103–103.

Juha-Pekka Tolvanen and Steven Kelly. 2009. MetaEdit+ defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 819–820.

Jon Whittle, John Hutchinson, and Mark Rouncefield. 2014. The state of practice in model-driven engineering. *IEEE software* 31, 3 (2014), 79–85.

PAPER E

ADDRESSING THE TRADE OFF BETWEEN SMELLS AND QUALITY WHEN REFACTORING CLASS DIAGRAMS

A. Barriga, L. Bettini, L. Iovino, A. Rutle and R. Heldal.

In Journal of Object Technology, Volume 20, Number 3, 2021.

JOT
JOURNAL OF
OBJECT TECHNOLOGY

Addressing the trade off between smells and quality when refactoring class diagrams

Angela Barriga[*], **Lorenzo Bettini**[†], **Ludovico Iovino**[‡], **Adrian Rutle**[*], **and Rogardt Heldal**[*]
[*]Western Norway University of Applied Sciences, Norway
[†]Università degli Studi di Firenze, Italy
[‡]Gran Sasso Science Institute, Italy

ABSTRACT Models are core artifacts of modern software engineering processes, and they are subject to evolution throughout their life cycle due to maintenance and to comply with new requirements as any other software artifact. Smells in modeling are indicators that something may be wrong within the model design. Removing the smells using refactoring usually has a positive effect on the general quality of the model. However, it could have a negative impact in some cases since it could destroy the quality wanted by stakeholders. PARMOREL is a framework that, using reinforcement learning, can automatically refactor models to comply with user preferences. The work presented in this paper extends PARMOREL to support smells detection and selective refactoring based on quality characteristics to assure only the refactoring with a positive impact is applied. We evaluated the approach on a large available public dataset to show that PARMOREL can decide which smells should be refactored to maintain and, even improve, the quality characteristics selected by the user.

KEYWORDS Smells, Refactoring, Quality evaluation, Reinforcement learning.

1. Introduction

Models are becoming core artifacts of modern software engineering processes (Whittle et al. 2014). Models, as happens with code, change and evolve throughout their life cycle due to maintenance and to comply with new requirements. Preserving the quality of these models is of the utmost importance to ease their maintenance and to correctly produce the systems they represent. To this extent, the model-driven engineering (MDE) community has developed a series of mechanisms to identify bad practices and smells that worsen models maintenance and to measure the quality of models.

Smells in code (Beck & Fowler 2018) are not bugs or errors but instead, can be considered as violations of the fundamentals of developing software that decrease the quality of code. In the same way, smells in modeling (Bettini et al. 2019) are indicators that something may be wrong within the model design, even if the model is valid. Some examples of domain modeling smells would be unnecessary duplicated features or classes isolated from the rest of the model, often resulting in uninstantiatable classes, especially if the model is instantiated with a class that could not reach the isolated one. Smells may severely affect the maintenance and evolution of models, as happens with code. Therefore, their early identification and removal is crucial to assure the final quality of models. There are many smells defined in the literature (Mumtaz et al. 2019; Beck & Fowler 2018; Strittmatter et al. 2016) and detecting and removing them is far from trivial. Refactoring models to remove smells might come with a cost. Since removing the smells imply modifying the model structure, this usually has a positive effect on the general quality of the model (Bettini et al. 2019) but, in some cases, it could also have a negative impact. This impact is strictly related to multiple aspects: the model's structural composition, smell

JOT reference format:
Angela Barriga, Lorenzo Bettini, Ludovico Iovino, Adrian Rutle, and Rogardt Heldal. *Addressing the trade off between smells and quality when refactoring class diagrams*. Journal of Object Technology. Vol. 19, No. 1, 2020. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2020.19.1.e1
JOT reference format:
Angela Barriga, Lorenzo Bettini, Ludovico Iovino, Adrian Rutle, and Rogardt Heldal. *Addressing the trade off between smells and quality when refactoring class diagrams*. Journal of Object Technology. Vol. 19, No. 1, 2020. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2020.19.1.e1

occurrences and combinations, etc.

However, to know the impact of the refactoring one needs a way to measure the quality of the model after the refactoring. In this paper, we will use quality characteristics. Quality characteristics have been extensively studied in the literature (Boehm et al. 1976; Dromey 1995; Ortega et al. 2003). With them, modelers can quantify how good models are in terms of concepts like analyzability, adaptability, understandability, etc. Several tools exist in code analysis and also in MDE where modelers can define their own characteristics and automatically detect them in models using various automated mechanisms (Basciani et al. 2019; López-Fernández et al. 2014).

The positive effect of calculating these quality characteristics automatically is that they can be used to measure the impact of removing a specific smell on the overall model or on specific quality characteristics (Di Rocco et al. 2014; García-Magariño et al. 2008). By combining the removal of smells and quality measurement, modelers could tackle the refactoring of models to remove smells without compromising the overall model quality, making it possible to find a balance between which smells should be removed and which ones not.

In our previous work (Barriga, Heldal, et al. 2020), we presented PARMOREL, an extensible model repair framework, implemented as an Eclipse plugin, which enables users to deal with different types of model issues and to add their own repair preferences to customize the results. This customization of results is achieved with reinforcement learning (RL) (Thrun & Littman 2000). By using RL, PARMOREL finds the best solution for repairing a model according to the user preferences. So far, as model issues, we tackled with PARMOREL the repair of syntactic errors in broken models. As user preferences, we have worked with quality characteristics (Iovino et al. 2020).

In this paper, we will demonstrate the flexibility of PARMOREL showing that it can support smells detection and refactoring. To achieve this, we integrate PARMOREL with a tool that allows modelers to identify smells and refactor them with known refactorings (Bettini et al. 2019). This extension is based on Edelta (Bettini et al. 2020), a DSL-based tool to define smells and corresponding refactorings in personalized libraries.

To validate this new extension, we solve the trade-off problem between smells and model quality in a dataset used in the literature, consisting of 404 class diagrams extracted from GitHub (Babur 2019). The results are encouraging and show that PARMOREL is able to decide which are the best smells to refactor in order to maintain and, even improve, the quality characteristics selected by the user.

Structure of the paper. This paper is organised as follows: Section 2 illustrates and presents the PARMOREL architecture. Section 3 demonstrates why we need to selectively remove smells instead of addressing all of them. Then, in Section 4, we show the customization applied to PARMOREL to perform selective removal of smells and how existing components have been extended, i.e., with Edelta and with a quality evaluation framework. In Section 5, we evaluate if PARMOREL can success in refactoring with a balance between smells and quality. Then, we present threats to validity in Section 6, explore the related work in Section 7 and conclude the paper in Section 8.

2. PARMOREL Framework

In this section, we briefly present the PARMOREL framework in order to understand its extension (in Section 4) to support selective refactoring of models containing smells. PARMOREL makes use of three main concepts: issues to be found in the models, actions to be applied in response to issues and preferences with which the user customizes how issues are solved. Then, a RL algorithm is in charge of deciding which is the best action to apply in response to an issue, according to preferences selected by the user. The architecture of PARMOREL is based on three main modules that we represented in Fig. 1: a *modeling module*, a *learning module* and a *preferences module*. In the rest of this section, we will explain these components and show how they make the framework flexible to be adapted to the needs of the users in terms of different models, issues, actions, preferences and learning algorithms.

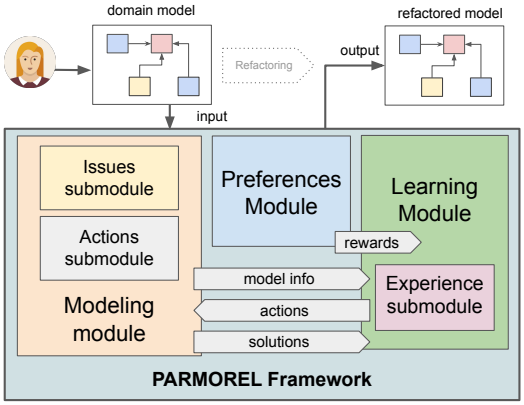


Figure 1 Overview of the PARMOREL architecture

2.1. Modeling module

The *modeling module* is divided in two submodules, namely the *issues submodule* and *actions submodule*.

The *issues submodule* is in charge of identifying which issues are present in the model and sends them to the *learning module*. In (Barriga, Heldal, et al. 2020) we introduced the concept of *issue*. An issue represents something that is improvable in a model regardless of its nature. An issue could be a syntactic or semantic error, a smell, a violation with respect to an architectural pattern or a specific constraint, etc.

The *actions submodule* is in charge of sending to the *learning module* the actions which are available for refactoring the model and of applying the chosen refactorings.

In this paper, we focus on extending the *modeling module* so that PARMOREL supports smell identification and refactoring. Therefore, more details about this module can be found in Section 4.

2.2. Learning module

The *learning module* makes use of RL to learn which actions are the best to refactor the issues in the models according to the preferences introduced by the users.

RL consists of algorithms able to learn by themselves how to interact in an environment without existing pre-labelled data,

only needing a set of available actions and rewards for each of these actions. RL allows PARMOREL to perform model manipulation without having any prior data (i.e., labelled data, historical data, etc.) about removing issues in models.

By using and tuning RL rewards, these algorithms can learn which are the best actions to apply to the model. RL rewards can be adapted to align with any preference introduced by the user as long as it can be quantified, e.g., improving quality characteristics (Iovino et al. 2020). Preferences need to be quantified so that their values can be mapped into RL rewards. For example, the value of the maintainability quality characteristic itself could be used as a reward, if the modeler wants to improve it.

Before finding a refactoring for a given model, PARMOREL is executed for a number of episodes. Each episode equals to one iteration refactoring the model. During the episodes, different actions will be applied to remove the different issues present in the model. For each of these episodes, a refactoring sequence is found, and by applying it, a provisional refactored model is created. The provisional refactored models are analyzed according to the preferences selected by the user, and the result is translated into rewards (e.g., the value of the considered quality characteristic of the refactored model). Hence, PARMOREL can identify how good the applied refactoring is according to the user requirements. Following this process, after each episode, actions leading to the results closest to the user requirements will have higher rewards and thus higher probabilities of being selected. After performing enough refactoring iterations, PARMOREL will select the refactoring with higher rewards and save the final refactored model.

RL is a broad field with many algorithms. In previous work, we compared the performance of different RL algorithms in PARMOREL and Q(λ) was the one that provided us the best performance (Barriga, Mandow, et al. 2020). Hence, we use Q(λ) in our current implementation.

Q(λ) In this algorithm, knowledge acquired is stored in a table structure called Q-Table (Thrun & Littman 2000). This table stores pairs of states (states equal smells in our application) and actions together with a Q-value. The Q-value is calculated using the rewards and it indicates how good each pair is. The Q-value is obtained with repeated calculations based on the Bellman Equation (Bellman 2013) as follows:

$$Q(s,a) = \alpha(r + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s,a)) \quad (1)$$

telling that the maximum future reward is the reward r the agent received for entering the current state s with some action a plus the maximum future reward for the next state s_{t+1} and action a' reduced by a discount factor γ.

This allows inferring the value of the current (s,a) pair based on the estimation of the next one (s_{t+1}, a'), which can be used to calculate an optimal policy to select actions. The factor α provides the learning rate, which determines how much new experience affects the Q-values. One of the variables used to calculate the Q-value, is the maximum weight stored in the Q-table for the next error to refactor ($\max_{a'} Q(s_{t+1}, a')$). This allows us to measure the consequences of applying a certain action in the model (e.g., if applying an action creates a new smell this action would be punished, getting a lower weight). At the end of the execution, pairs with the highest Q-value will conform to the policy to solve the problem. Our algorithm is epsilon-greedy (ε-greedy): it avoids local optima using an exploration-exploitation trade-off by exploring (i.e. choosing a random action) with probability ε, and exploiting (i.e. choosing the action with highest Q-value) the remainder of the time. According to our testing (Barriga, Mandow, et al. 2020), we obtain better results with an ε of 0.3. Regarding other parameters, discount factor (γ), and learning rate (α), we use 1.0 for both of them.

Q(λ) uses a technique called *eligibility traces* (see lines 9-18 in Algorithm 1) to back-propagate the values and received rewards, but it does so not only to the immediately preceding state *e(s,a)* (or pair of state-action), but to all preceding states of the current episode, (stored in the *sae* list, see lines 16-18). The idea is that this propagation decays in intensity the further a state is in the past. This decayed propagation can lead to a speed up in the algorithm's convergence, especially in sparse reward models (Thrun & Littman 2000), which provides rewards only at the end of each episode (e.g., PARMOREL receives the quality characteristics rewards from the provisional refactored model at the end of an episode). The propagation decay is controlled with a parameter λ (see line 18). In practice, the speed of convergence as a function of the value of λ (between 0 and 1) generally has a U-shape. Therefore, the optimal convergence is usually achieved with an intermediate value of λ, which needs to be determined experimentally. According to our experiments (Barriga, Mandow, et al. 2020), we get the best results by giving λ a value of 0.7. Lower or higher values lead to results of lower quality. The new Q-value is temporarily stored in the variable δ (see line 15). It is later stored in the Q-table (see line 17) by adding the already stored Q-value for that pair of state-action (s,a) to the product of α, δ (the new Q-value) and the eligibility trace of (s,a).

The pseudocode depicted in Algorithm 1 is adapted from the one presented in chapter 12 in (Thrun & Littman 2000).

Algorithm 1 Q(λ)

```
1:  Initialize Q-Table
2:  for each episode do
3:      Initialize eligibility table e (default value 0)
4:      Initialize sae as an empty list of state-action pairs
5:      s ← initial state s_0
6:      while errors in model != ∅ do
7:          Get state s
8:          Select best action a with ε-greedy policy for s
9:          if a is selected randomly then
10:             reset eligibility to 0
11:             reset sae as an empty list
12:         s_{t+1} ← a applied in s
13:         Add (s,a) to sae list
14:         e(s,a) ← e(s,a) + 1
15:         δ = r + γmax_{a'} Q(s_{t+1}, a') − Q(s,a)
16:         for each s,a in sae do
17:             Q(s,a) = Q(s,a) + αδe(s,a)
18:             e(s,a) ← γλe(s,a)
19:         t ← t + 1
20:         s ← s_{t+1}
```

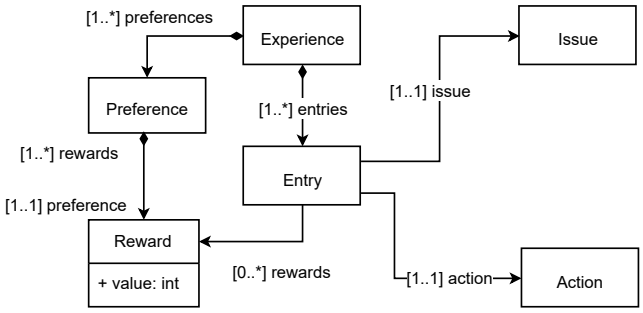


Figure 2 Model of experience in PARMOREL

Experience submodule One of the advantages of using RL is that these algorithms can improve their performance the more they are applied. In our approach, the more PARMOREL modifies models, the better performance it might get. This is because PARMOREL acquires and builds experience that is reused in later refactorings. To this end, we define the *experience submodule*. This submodule makes use of the machine learning (ML) technique of transfer learning (TL) (Barriga, Rutle, & Heldal 2020). In traditional RL, the value of each pair of issues and actions depends on a single reward; e.g., for a robot learning how to escape a maze, it receives a negative reward when stepping into a wall and a positive one when entering a free space. However, in our case one pair's weight may depend on multiple rewards since it might involve several user preferences, e.g., a user might want to boost the maintainability and reusability of a model. Introducing user preferences complicates reusing the experience acquired by the RL algorithm, since what is a good refactoring for one user might not be acceptable for another one. With this technique, what is learnt from the refactoring of one model could be reused for other models. Hence, consequent executions of PARMOREL could achieve better performance the more experience is reused. Even if the users are different, if the preferences they selected and the issues present in the models are similar, sharing experience would be useful.

We use the model in Fig. 2 to illustrate how PARMOREL supports TL. The learning information gained after each refactoring is represented by the concept Experience which is composed of one to many entries and preferences. The concept Entry refers to the pairs in the Q-table and hence it has references to all the elements that are part of the Q-table: an Issue and an Action. In addition, an Entry has a zero to many references to Reward. The Reward contains a numerical value based on the users' preferences.

The rewards stored in the Experience are used to initialize the Q-table in following executions. This way, if the current user shares any preference with previous ones, the rewards these previous preferences provided in previous refactorings can be used to initialize the new user's Q-table, so that the refactoring does not start from zero. This way, the learning will converge faster and less episodes will be required. When sharing experience in PARMOREL, we reduce the value of ε (see line 8 in Algorithm 1) from 0.3 to 0.15 to enhance the influence of the previous Experience. We initialize the Q-table with the accumulated rewards of the shared preferences multiplied by a

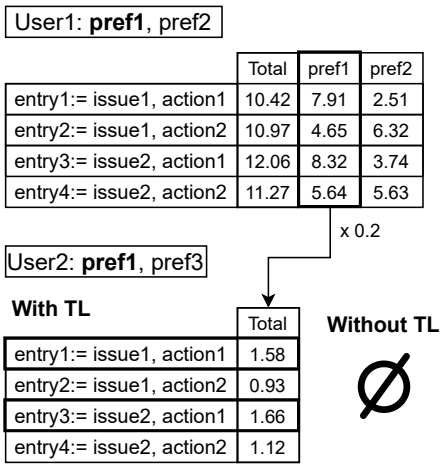


Figure 3 TL between 2 users with a shared preference

discount factor of 0.2. This way we assure previous refactoring processes influence the new ones by jump-starting the process but without interfering with learning new refactoring sequences. Based on our experimental results (Barriga, Rutle, & Heldal 2020), we found that a value of 0.2 gave the best results for our cases. This parameter's value can be modified to affect the impact of previous experience on new refactorings. However, the value should remain a constant during the execution otherwise some parts of the experience will be more favoured than others.

An example of this process is displayed in Fig. 3. In the left part of the image we show the Q-table of *User1* once she finishes using PARMOREL. *User1* chooses as preferences *pref1* and *pref2* to refactor a model with two issues, namely *issue1* and *issue2*. Both issues can be refactored with actions *action1* and *action2*. Then, in the right part of Fig. 3 we show how the Q-table will look for *User2* once she starts using PARMOREL. This user chooses to refactor with preferences *pref1* and *pref3*. The model to refactor is different than the one refactored by *User1*, but since what is relevant for PARMOREL are issues and actions, the *Experience* can be reused regardless of the specific model to refactor. Without TL the Q-table will not exist and a new one will be created, adding more time to the processing part of the learning algorithm. With TL, every entry existent in the *Experience* is copied in the Q-table, and since *pref1* is shared with *User1*, the Q-table is initialized with the rewards provided from this preference multiplied by the discount factor. This way, when PARMOREL starts the refactoring process for *User2*, the time spent in populating the Q-table is reduced and the learning algorithm will already have an intuition of which actions are better for each issue.

For more details about how PARMOREL uses TL and how the *experience submodule* works we refer the reader to our previous work (Barriga, Rutle, & Heldal 2020; Barriga, Heldal, et al. 2020).

2.3. Preferences module

Users can customize the results PARMOREL produces with their own preferences. PARMOREL supports preferences as long as they can be translated into numeric values. PARMOREL will take these values as rewards that will guide the refactor

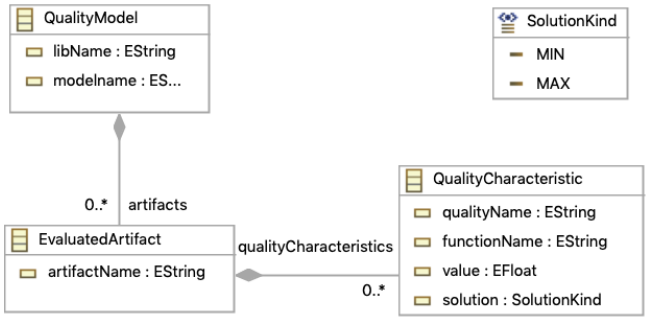


Figure 4 Quality characteristics model

process.

For example, users could prefer to refactor improving a quality characteristic (e.g., maintainability, reusability, understandability, etc.), to minimize the model distance with respect to the original model, etc. PARMOREL will use the rewards to estimate how good or bad each action is to satisfy the user preferences. As part of the preferences given to the users, PARMOREL integrates a quality evaluation tool (Iovino et al. 2020), which is inspired by (Basciani et al. 2016).

Quality Characteristics as preferences This quality evaluation tool supports the specification of quality characteristics conforming to the domain model in Fig. 4. Each EvaluatedArtifact (the artifact from which the quality characteristics will be measured, e.g.; a domain model) will be assigned a set of QualityCharacteristics which can be specified by the modeler. Moreover, whether quality characteristics should be maximized or minimized, is specified in the attribute solution. The calculation function functionName of each quality characteristic has to match with a definition of an EOL (Kolovos et al. 2006) script aggregating the available metrics (as shown in the various formulas) in a predefined library (Basciani et al. 2019). EOL (Kolovos et al. 2006) is an imperative programming language for creating, querying and modifying EMF models. EOL offers model management operations with a dedicated language built on top of EMF. This makes it easier to define evaluation operations compared to Java implementations using the EMF API directly (Basciani et al. 2016).

In this paper, we specify the following quality characteristics to be used as user preferences: maintainability, understandability, complexity, and reusability.

The *maintainability* has been defined according to the definition given in (Genero & Piattini 2001) and the formula presented in (Basciani et al. 2016), that is based on some of the metrics shown in Table 1 as follows:

$$Maintainability = \left(\frac{NC + NA + NR + DIT_{Max} + Fanout_{Max}}{5} \right) \quad (2)$$

The definitions of the *understandability* and *complexity* quality characteristics are adopted from (Sheldon & Chung 2006). In particular, *understandability* can be defined as follows:

$$Understandability = \left(\frac{\sum_{k=1}^{NC} PRED + 1}{NC} \right) \quad (3)$$

Characteristic	Acronym
Number of classes	NC
Number of references	NR
Number of opposite references	NOPR
Number of containment references	NCR
Number of attributes	NA
Number of unidirectional references	NUR
Max. generalization hierarchical level	DITmax
Max. reference sibling	FANOUTmax
Number of features	NTF
Sum of inherited structural features	INHF
Attribute inheritance factor	AIF
Number of predecessor in hierarchy	PRED

Table 1 Metrics used in the quality characteristics equations

where PRED regards the predecessors of each class, since, in order to understand a class, we have to understand all of the ancestor classes that affect the class as well as the class itself.

Complexity can be defined in terms of the number of static relationships between the classes (i.e., number of references). The complexity of the association and aggregation relationships is counted as the number of direct connections, whereas the generalization relationship is counted as the number of all the ancestor and descendant classes. Thus, the *complexity* quality characteristic can be defined as follows:

$$Complexity = (NR - NUR + NOPR + UND + (NR - NCR)) \quad (4)$$

where NUR is the number of unidirectional references measured as the difference between bidirectional and number of references, and UND is the *understandability* value measured as defined in Equation 3.

The *reusability* of a given model can be measured in different ways. One of these is to use the attribute inheritance factor *AIF* as proposed in (Arendt & Taentzer 2013). As presented in (Al-Jáafer & Sabri 2007), AIF can be defined as follows:

$$Reusability = AIF = \left(\frac{INHF}{NTF} \right) \quad (5)$$

where *INHF* is the sum of the inherited features in all classes, and *NTF* is the total number of available features.

3. Dilemma: Removing all the smells?

According to (Arendt & Taentzer 2013) a model quality assurance framework should implement three important iterative phases: i) model analysis, ii) identification of smells and iii) removing of the smells. In order to confirm that removing the smells had a positive effect not only formally but also practically, quality evaluation is crucial and can be considered as a litmus test of the refactoring activity. For this reason, it is helpful to evaluate the model before and after removing the smells to see if the applied refactorings effectively improved the design of the model.

In this section, we use an explanatory example to motivate our research. We demonstrate that removing all the smells in a model may be beneficial in terms of quality, but we can have

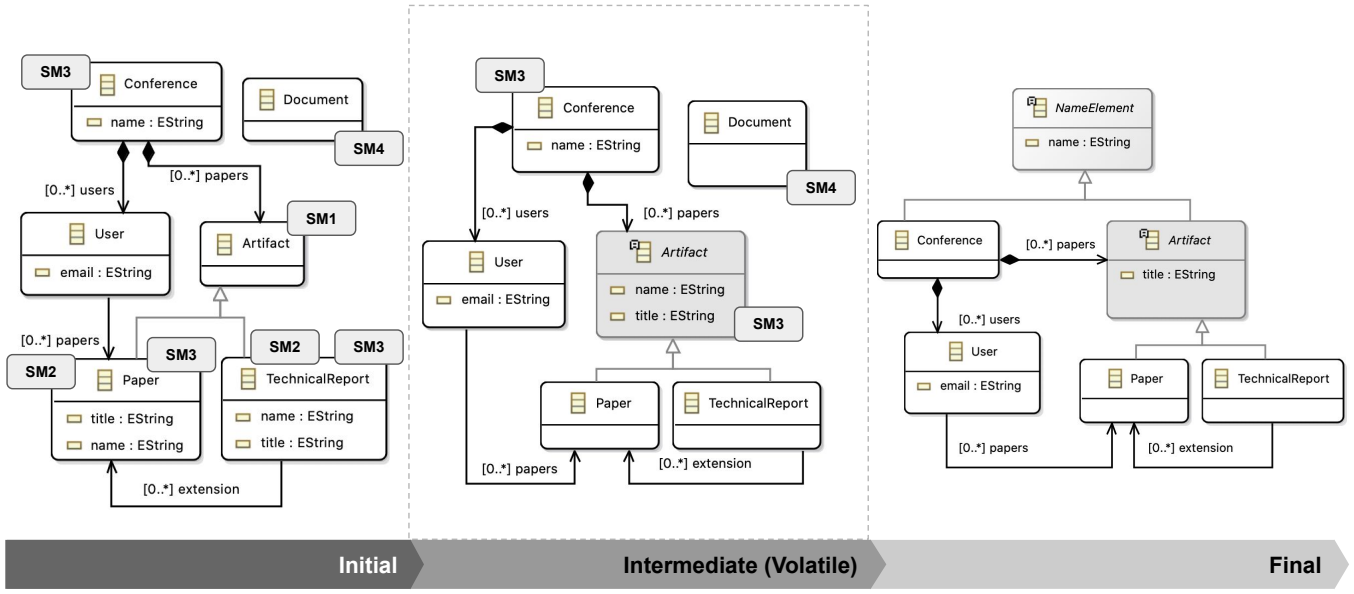


Figure 5 Running example showing an initial smelly model, an intermediate version of the model where SM1 and SM2 are removed, and a final version where SM3 and SM4 are removed

cases in which not all the quality characteristics improve. These cases strictly depend on the model containing the smells, number of occurrences and the structure of the parts that are not affected by the smells. For instance, evaluating the quality of a huge model with only one smell can give very different results with respect to a small model containing the same smell. Moreover, certain type of smells may affect specific quality characteristics because of the parts of the model they affect (Strittmatter et al. 2016; Basciani et al. 2016; Bettini et al. 2019), as we will see later in this section.

Removing all the smells unconditionally should improve the quality characteristics, but depending on the model structure, smell occurrences and applied refactorings, some of the quality characteristics may get worse. In Fig. 5 we introduce, as an example, a "smelly" model. This domain model is inspired by an example taken from the ATL Zoo and it represents a simplified conference management system that can be used internally by universities or departments. We use this trivial example to highlight the issues and motivate the problem, whereas in Section 5 we will show case studies part of a real dataset.

In this system, as can be seen from Fig. 5 (initial), the modeler can declare a Conference, that contains the submitted Artifacts, that can be identified with a title and a name. Moreover, a set of Users can be defined and registered to the system with their email addresses. In particular, users can be assigned to papers. A Conference contains a set of Artifacts which might be either a Paper or a TechnicalReport, that may extend one or more papers. These two classes extend the class Artifact which is declared as concrete. The possibility of an accidental instantiation of the class Artifact leads to the smell *concrete abstract class* (SM1). Three of the classes in this model, Conference, Paper and TechnicalReport, share the attribute name, which can be identified with the smell SM3, *duplicated features*. The two subclasses of Artifact (i.e., Paper and TechnicalReport) share the attributes title and name (String). This identifies the smell *duplicated features in hierarchy*, i.e., SM2, which is a more specific version of *duplicated features*: here the same feature is found in all of the subclasses of a given superclass. Finally the class Document is declared, maybe with a missing relationship to any of the other classes. This implies the *dead class* smell, i.e., SM4. It is worth noting that the class Document could not be instantiated in a framework like EMF—due to EMF's requirement that all model elements should be contained in a root model element—since Conference is the root of our model. One way to instantiate it is to declare it as root of the model but then the modeler would not be able to instantiate the remaining classes of the metamodel, hence the class is identified as dead.

These four smells may affect multiple quality characteristics, and when multiple smells are automatically removed with refactorings, the quality characteristics can improve but in some cases can also get worse. In the case reported in Fig. 5, four possible refactorings may be applied:

SM1 Concrete abstract class → Make the class abstract
SM2 Duplicated features in hierarchy → Pull up features
SM3 Duplicated features → Extract superclass
SM4 Dead class → Remove class

When our smell finder detects a smell, the corresponding refactoring is immediately applied. Applying all the refactorings immediately after finding the smells might lead to models with lower quality characteristics than the original smelly models. In Fig. 5, the intermediate model shows the model after applying the refactoring for SM1 (Artifact is made abstract) and SM2 (name and title are pulled up into Artifact), whereas the final model in Fig. 5 shows the model after applying all the refactorings, including also a newly created instance of SM3. Hence a new class NameElement is added as superclass for

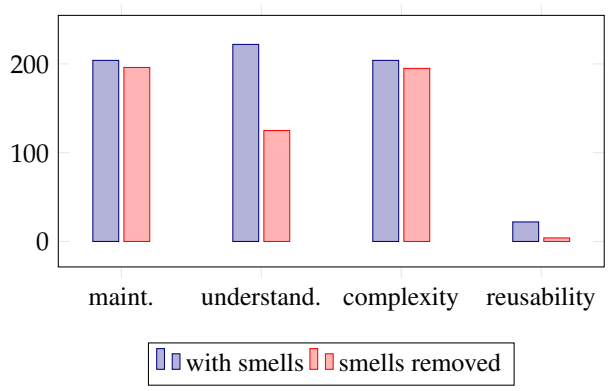


Figure 6 The quality characteristics maintainability (maint.), understandability (understand.), complexity, and reusability before and after removing all the smells in the model in Fig. 5

Conference and Artifact) and SM4 is removed by deleting the dead class Document. Although we identified both SM2 and SM3 on the classes Paper and TechnicalReport, we removed the more specific smell SM2. This strategy incorporated in our smells finder and resolver (see Section 4.1) makes sense with respect to how a modeler would refactor this smell, however, it would at the end produce the final model in Fig. 5 which is of lower quality than the original smelly model. Figure 6 displays some quality characteristics, namely, maintainability, understandability, complexity and reusability, measured in the final model before and after the smells are removed. That is, by removing all the smells automatically with the listed refactorings, we will get worse values in all these characteristics except for reusability.

In our example, applying the refactoring associated with SM3 extract superclass, worsen the overall quality of the model, since it affects the maintainability, understandability and complexity by adding a new element into the model. However, SM3 improves the reusability, since it creates more inherited features. The refactorings associated with SM2-SM3 improve the reusability and maintainability by removing features from the model. Removing classes to solve SM4 worsens understandability and complexity while it improves maintainability. Regarding SM1, removing it does not affect the quality characteristics considered in this example, however, if unsolved, it could deteriorate the model's quality in the future, since a class that is concrete when it should be abstract could be incorrectly instantiated.

By combining these refactorings we could face situations where removing several smells lead to no improvement in the model quality, for example by removing SM1 and SM4 we get worse understandability and complexity, without improving any quality characteristics. This is an indication that, by selectively applying refactorings when removing smells, quality characteristics could improve with respect to automatically removing every smell in a model.

To overcome these limitations and complement the automatic approaches as (Bettini et al. 2019; Arendt & Taentzer 2013) we propose a new application of PARMOREL to find a balance between smells refactoring and models quality.

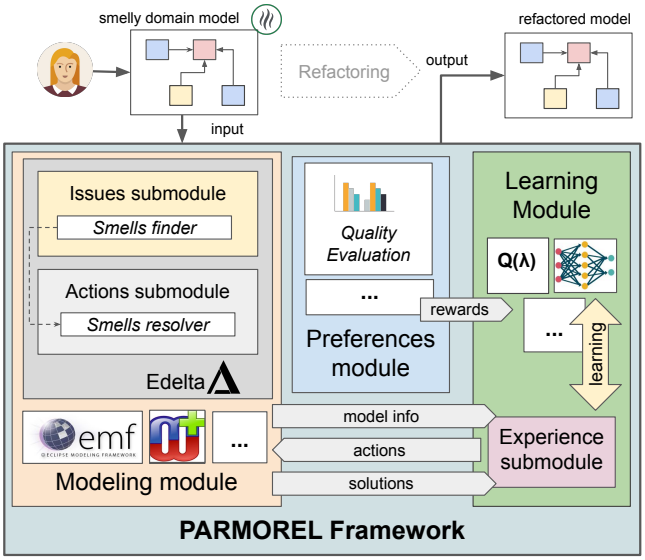


Figure 7 Detailed architecture of the framework

4. Selective smell removal

In previous work (Barriga, Rutle, & Heldal 2020; Barriga, Heldal, et al. 2020; Iovino et al. 2020), we have applied PARMOREL to repair faulty models that violate certain constraints of the Ecore metamodel. In order to apply it for refactoring smells, there are some parts of the framework that must be adapted. In this section, we detail how the PARMOREL framework has been extended to support this task. Figure 7 displays the architecture of the framework in detail after this extension.

4.1. Modeling module extension

In this paper, we extend the modeling module to identify and refactor smells by using EMF (Steinberg et al. 2008) together with Edelta (Bettini et al. 2017) for refactoring Ecore models.

Edelta Edelta is a model refactoring tool, based on a DSL, for easily defining Ecore model evolutions and refactorings. The core features of Edelta and its DSL have been detailed in (Bettini et al. 2017). Edelta provides modelers with constructs for specifying atomic evolutions and complex refactorings. Atomic evolutions are simple changes applied to models, i.e., additions, deletions and edits. Complex refactorings are reusable changes, defined by composing already defined atomic or complex refactoriings. The Edelta DSL has been implemented with Xtext (Bettini 2016) and also a complete IDE based on Eclipse is available, offering syntax highlighting, code completion, error reporting, incremental building, as well as debugging. Recently, in (Bettini et al. 2020), the new version of Edelta was presented, supporting a completely live environment, where the modeler can have an immediate feedback in the IDE of the evolved Ecore models. Edelta has also been used for detecting Ecore model smells and for removing them by means of reusable refactorings organised in libraries (Bettini et al. 2019). In previous work, Edelta has been used as a standalone tool that can be used on a subject metamodel to analyze it, evolve it or to apply refactorings, interactively, with the live IDE environment of Edelta.

Moreover, all these mechanisms can also be used in a standard Java program to process a set of metamodels in batch mode.

In this paper, we make use of Edelta libraries to instantiate the *issues submodule* and the *actions submodule* with a *smells finder* and *smells resolver*, respectively.

Smells finder The *smells finder* uses the Edelta DSL to specify queries for identifying smells in Ecore models. Using the Edelta language, the modeler can provide the specification of custom smell finders and refactorings, which can be properly organized in reusable libraries. The Edelta DSL has a Java-like syntax, so it should be easily understood by Java programmers, but with less "syntactic noise". For example, most of types declarations can be omitted if they can be inferred from the context. Moreover, the Edelta DSL is based on the Java type system and it is completely interoperable with all existing Java types and libraries. Indeed, the types used in the next listings are Java types.

Edelta comes with a smell finder including the smells mentioned in this paper, but the modeler can further extend this library with new smells or refine the existing ones. In Listing 1 we report an extract of an Edelta library containing a few smells finders mentioned in Section 3.

```
def findDuplicatedFeatures(EPackage epackage) {
 return findDuplicatedFeaturesInCollection(
  ePackage.allEStructuralFeatures,
  [existing, current| new EdeltaFeatureEqualityHelper()
   .equals(existing, current) ] )
}
def findDuplicatedFeaturesInCollection(
    Collection<EStructuralFeature> features,
    BiPredicate<EStructuralFeature, EStructuralFeature>
     matcher) {
 val map = newLinkedHashMap
 for (f : features) {
  val existing = map.entrySet.findFirst[matcher.test(it.key
     , f)]
  if (existing != null) {
   existing.value += f
  } else {
   map.put(f, newArrayList(f))
  }
 }
 return map.filter[key, values| values.size > 1]
}
def findConcreteAbstractMetaclasses(EPackage ePackage) {
 return ePackage.allEClasses
  .filter[cl | !cl.abstract && cl.hasSubclasses]
}
...
```

Listing 1 Edelta snippet of the smell finder library

Concerning finding duplicated features, the core function is `findDuplicatedFeaturesInCollection`. This operation takes the collection of features to inspect and a lambda expression[1] that is responsible of deciding whether two features should be considered equal in two different classes[2]. Both `findDuplicatedFeaturesInHierarchy` and `findDuplicatedFeatures` call this operation with a different collection of features to inspect and with a lambda expression that relies on our default implementation of equality detection for features, which scans all the properties of two given features. Note that modelers can reuse `findDuplicatedFeaturesInCollection` with a custom equality matcher for their own new smells and refactorings. The smell finder returns the possible detected duplicated features in an appropriate data structure (in this case, a map). Such a data structure contains the information needed to possibly "resolve" the smell, as shown in the next paragraphs. The definition of `findConcreteAbstractMetaclasses` should be straightforward. In the above code we rely on some utility functions defined in Edelta (e.g., `allEClasses`, `directSubclasses`, etc.) that we do not detail here.

The *smells finder* implements the *issues submodule* in the PARMOREL framework and hence, it takes care of identifying which smells are present in the models and communicating them to the *learning module*.

Smells resolver To complement the *smells finder* we use a *smells resolver* to remove the smells found in the models. When a smell is declared the modeler needs to specify the refactoring to resolve the smell. This correspondence is declared in a Edelta library called *resolver* that basically links the smells with the refactorings. Model refactorings are specified by using the Edelta DSL as well. For instance we could specify that the *duplicated features in hierarchy* should be resolved by *pull up attributes*, that the more general *duplicated features* smell should be resolved by *extract superclass*, and that the *concrete abstract class* should be resolved by simply making the class abstract.

An important feature of Edelta is that it resolves all occurrences of a smell type in one run. For instance, when resolving SM2 in Fig. 5, both of the attributes name and title are pulled up to the same class Artifact. The impact of this batch resolution would be more visible if we had two common superclasses for Paper and TechnicalReport, since in this case, atomic resolutions would lead to potentially pulling up name to one of the superclasses and title to the other one. Listing 3 reports a few Edelta refactorings that we have defined in the catalog published at https://www.metamodelrefactoring.org that we used in the above resolver functions in Listing 2. In particular, in this Listing we show the Edelta operations for *pull up* and *extract superclass* (functions like `addNewEClass` and `addEStructuralFeature` are examples of Edelta atomic refactorings).

```
def resolveDuplicatedFeaturesInHierarchy(EPackage pack) {
 finder.findDuplicatedFeaturesInHierarchy(pack)
  .forEach[superClass, duplicates|
   duplicates.forEach[key, values|
    refactorings.pullUpFeatures(superClass, values) ] ]
}
def resolveDuplicatedFeatures(EPackage pack) {
 finder.findDuplicatedFeatures(pack).values
  .forEach[refactorings.extractSuperclass(it)]
}
def resolveAbstractSubclassesOfConcreteSuperclasses(
     EPackage pack) {
 finder.findAbstractSubclassesOfConcreteSuperclasses(pack)
  .forEach[makeConcrete]
}
...
```

Listing 2 Edelta snippet (i) of the resolver library

[1] In Edelta lambda expressions have the shape: `[ param1, param2, ... | body ]`. As in Java, types of parameters can be omitted when they can be inferred. Note that the lambda expression is assignable to the Java functional interface `BiPredicate`.

[2] We do not report all the code since the complete implementation of the smell finders can be found in the source files of the Edelta plugin: https://github.com/LorenzoBettini/edelta.

```
def extractSuperclass(List<? extends EStructuralFeature>
    duplicates) {
 val feature = duplicates.head;
 val name = feature.name.toFirstUpper + "Element";
 val containingEPackage = feature.EContainingClass.EPackage

 containingEPackage.addNewEClass(name) [
  makeAbstract
  duplicates.map[EContainingClass].forEach[c | c.
     addESuperType(it)]
  pullUpFeatures(duplicates)
 ]
}
def pullUpFeatures(EClass dest, List<? extends
     EStructuralFeature> duplicates) {
 duplicates.head.copyTo(dest)
 removeAllElements(duplicates)
}
...
```

Listing 3 Edelta snippet (ii) of the resolver library

The Edelta DSL supports the definition of new smell finders and refactorings that can be coupled together and thus creating new resolvers (as can be seen in Listing 2). These finder-resolver pairs can be organized in a way which dictates the order in which the smells are resolved by Edelta. Although this order is important, the modeler could also specify mutually-exclusive smell finders since the Edelta specification allows for user-defined libraries. For instance, one could define *duplicated features not in hierarchy* as a counterpart for *duplicated features in hierarchy* so that model elements matched by a former smell finder are not related with model elements matched by a latter one. In this way, an implicit order of resolutions could be defined. All such new smell and resolver definitions are automatically available to the entire ecosystem.

The *smells resolver* implements the *actions submodule*, so it notifies the *learning module* about the refactorings available for each smell. Additionally, it applies the chosen refactorings in the model. As mentioned in Section 3, the integration with an external tool like PARMOREL could also reuse the order of invocation of the resolvers. Since PARMOREL removes smells by their types (e.g. all instances of SM2) we rely on the order in which the smells are found and resolved.

4.2. Issues

Previously, we tackled issues individually. PARMOREL would address them one by one regardless of their type or possible duplicities. This made sense since, in a broken model with syntactic errors, the desirable solution is that all errors are removed. Additionally, these errors could have multiple potential solutions that could modify drastically the model structure.

In our current scenario, we contemplate the possibility to leave smells unsolved as long as this is beneficial for the overall quality of the model. Now, for each smell type, the *smell resolver* provides us with one possible solution. Hence, PARMOREL has to learn whether it is worth it or not to apply the refactoring. According to our testing with the smell types and their refactorings which are implemented for this paper (see Section 5), removing a particular smell type will have a very similar impact on the quality characteristics of the models. This is because the quality characteristics we consider are based on the number of different elements in the models. Changing the number of specific elements with addition or removal or setting values, will affect some quality characteristics positively and others negatively.

Because of this, we consider smells in batches, organized by their types. For example, if a model present 3 instances of SM1 (see Section 3), PARMOREL will tackle SM1 as a batch, deciding to refactor or leave it unsolved, instead of tackling the 3 instances individually.

4.3. Episodes

With this batch organization, we reduce the time needed for refactoring, since the maximum number of episodes the RL algorithm will run depends on the found smell types, and not the smell instances.

As explained in Section 2, an episode equals to one iteration refactoring the model. In each episode, a possible refactoring sequence is found, and by applying it, a provisional refactored model is created. At the end of all the episodes, PARMOREL will have learned which are the best actions to solve the issues in the model according to the user preferences. The maximum number of episodes which PARMOREL runs is a parameter within the framework.

According to our testing, PARMOREL needs, for learning the best refactoring for each model, a maximum of 50 episodes for each present smell type. The more smell types a model contains, the longer PARMOREL will require to learn which is the best refactoring for it. For example, for a model with one smell type, PARMOREL will require a maximum of 50 episodes to converge, while for a model with 5 smell types, the maximum will be 250.

To avoid reaching the maximum needlessly, we run the RL algorithm with an early-stopping criteria. The learning will stop once $max_a Q(s_0, a)$ (the maximum Q-value of the initial state) remains unchanged for 25 episodes.

4.4. Rewards

To support the combination of several quality characteristics as a preference, it is not enough to directly use the values of the characteristics as a reward.

According to the characteristics definitions presented in Section 3, maintainability, understandability and complexity are decreasing characteristics. This means that lower values in these characteristics are an indicator of better quality. By contrast, reusability is an increasing characteristic, meaning that the higher its value is, the better reusability the model has. Hence, we could directly use increasing characteristics values, but decreasing ones need to be converted so that their values can be used as a reward.

For example, a user wants to improve the maintainability and reusability characteristics of a model which initial values (v_0) are 10 and 0.15, respectively. For this model, PARMOREL finds two possible refactorings, R1 and R2, each leading to the following quality values (v_r): R1: maintainability of 9.6 and reusability of 0.02 and R2: maintainability of 9.2 and reusability of 0.17. Maintainability improves in both refactorings while reusability gets better in R2 and worse in R1. If we directly added these values we would obtain a reward of 9.62 for the first refactoring and 9.37 for the second one. With this, PARMOREL

would choose R1 although it worsens reusability rather than choosing R2 which improves both characteristics and gives a better result in maintainability.

To avoid this situation, for every decreasing characteristic, we subtract v_r from v_0 and add v_0 back to the result (see Equation 6). With this, we convert the characteristics values so that the higher they are, the better quality they imply.

There could be situations where different quality characteristics have very different ranges. To avoid that one of the characteristics has more influence on the reward than the others, we transform the values v so that they reflect the improvement each characteristic has undergone within a closer range (see the value x in Equation 7). For example, by applying Equation 6 for R2, where v_r values are 9.2 (decreasing) and 0.17 (increasing), and the v_0 values are 10 and 0.15, respectively, we obtain the v values 10.8 and 0.17. Applying Equation 7 to these values, we obtain the x values 108 and 113.3. Finally, by applying Equation 8, (where n is the number of quality characteristics selected by the user), we add all x values and we obtain the *reward*.

By doing this, the example refactorings would get a reward of 117.3 for R1 and 221.3 for R2. Hence, PARMOREL would choose R2. To make them easier to read, values in Fig. 6 were converted so that higher values imply higher quality by using these equations.

$$v = \begin{cases} (v_0 - v_r) + v_0, & \text{if decreasing characteristic} \\ v_r, & \text{if increasing characteristic} \end{cases} \quad (6)$$

$$x = \frac{v * 100}{v_0}, \quad \text{if } v == 0 \text{ then } x = 0 \quad (7)$$

$$reward = \sum_{i=1}^{n} x_i \quad (8)$$

5. Evaluation

In this section, we present an evaluation of the proposed approach. In particular, we aim at answering the following research question:

> ***RQ:*** *How well can PARMOREL refactor models with a balance between removing smells and at the same time improving their quality?*

Experiment setup and dataset In this paper, we consider the following quality characteristics (Genero & Piattini 2001): maintainability, understandability, complexity, and reusability. These characteristics are offered to PARMOREL users as preferences. In this experiment, as user preferences, we select to improve maintainability and reusability. These preferences are mapped into reward values as explained in Section 4.4. These characteristics are opposite and, usually, when one improves the other worsens. This adds more challenge to the evaluation, since PARMOREL needs to find a balance for satisfying both quality criteria at the same time. For this evaluation, we choose the dataset from (Babur 2019), containing 555 Ecore models extracted from GitHub. We run PARMOREL in Eclipse 2020-06 (the Modeling package) on a laptop with the following specifications: Windows 10 Home, Intel Core i5-6300U @2.4GHz, 64 bits, 16GB RAM.

Smell	Refactoring
1-Concrete abstract class	Make the class abstract
2-Duplicated features in hierarchy	Pull up features
3-Duplicated features in classes	Extract superclass
4-Dead class	Remove dead class
5-Redundant container relation	Set correct reference as opposite
6-Abstract subclasses of concrete superclass	Make subclasses concrete
7-Abstract concrete class	Make the class concrete
8-Classification by hierarchy	Transform the hierarchy to enum

Table 2 Smells and refactorings supported in the evaluation

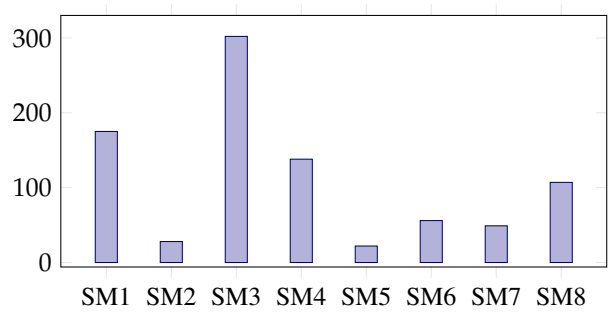


Figure 8 Distribution of smells throughout the dataset

Only 58 models in the dataset do not contain any smell, meaning that 89.54% of the models present some type of smell. From the models with smells, we discard 93, since they are not supported by the quality evaluation tool and hence we can not extract their quality and use it as rewards in the RL algorithm. This makes a total of 404 models subject to be refactored.

The models are of diverse size, containing between 10 and 445 elements, counting classes, attributes and references. From the 8 smell types we have defined in the *smells finder* for this evaluation, each model present between 1 and 7 types. Counting individual instances of each smell type, the models present between 1 and 52 smell instances. Table 2 details the defined smells and the refactoring for each of them. Figure 8 shows the number of models in the dataset containing each smell type.

We randomly split the dataset of models with an 80-20% distribution, refactoring 20% of the models twice, with and without having first refactored the 80%. With this, we analyze the impact of reusing learning with the *experience submodule* on the refactoring time of the 20%.

Analysis of results When refactoring the 80+20% of the dataset, it takes PARMOREL between 0.9 and 56.5s to learn how to refactor each model.

When refactoring the 20% independently, without reusing learning, it takes PARMOREL an average of 37% more time to refactor these models. Faster refactoring happens in models with bigger size, since the bigger the models, the more learning can be reused from previous refactorings. By comparing the refactor time from refactoring with and without reusing learning, we can conclude that PARMOREL streamlines the refactor time of the models between 2% and 61% when it has learned from refactoring other models.

Regarding maintainability, PARMOREL is able to improve it in 33.6% of the models. For 40% of the models it remains unchanged and, for the remaining 26.3%, it worsens. For reusability, 74.50% of the models present better results after refactoring and 25.2% remains unchanged. Only one model from the dataset presented worse reusability after refactoring. These results are summarized in Fig. 9. In 100% of the models, whenever one of the characteristics worsened, the other one improved. These refactorings were selected because in the trade-offs between the characteristics values the best decision for the overall quality of the model was to worsen one of the characteristics in benefit of the other one.

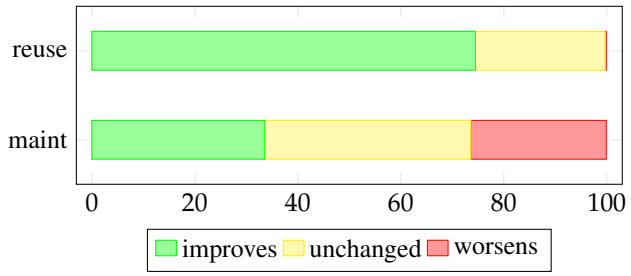


Figure 9 Reusability and maintainability results

Figure 10 displays the percentage of each smell type removed from the total present in the models for which maintainability and reusability improves, respectively. SM1-3 are mostly removed in both cases, while SM4-8 are mostly ignored. Moreover, SM4 and SM8 are more often removed when reusability improves, mostly because the refactoring of these smells reduce the total number of elements in the model.

Taking into account the characteristics in combination, PARMOREL was able to improve the quality of both of them in 31.43% of the models in the dataset. It also improves one of the two characteristics in 45.29% of the models while for 23.28% both remained unchanged (see Fig. 11).

As a conclusion, only in 22.27% of the models the best solution found by PARMOREL was to remove all the smells With the results of this evaluation, we can conclude that when taking into account the quality of the models, the best solution is usually not to remove all the smells. Hence, as an answer to

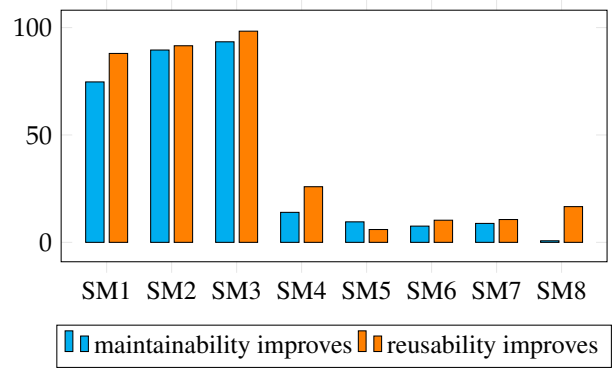


Figure 10 Percentage of each smell type fixed when quality improves

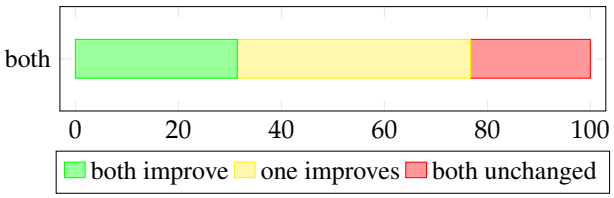


Figure 11 Both characteristics results after refactoring

our research question, PARMOREL is able to refactor the models with a balance between which smells should be addressed without degrading the quality of the models and even improving it. In most cases, the refactored model presents higher quality in the characteristics selected by the user than the original one (76.72% of the models in the evaluation). Additionally, as Fig. 10 shows, PARMOREL has the tendency to remove some of the smell types and to ignore others.

6. Threats to validity

In this section, we discuss potential threats that are associated with the validity of the experiments discussed in Section 5. We distinguish between internal and external threats to validity as in the following:

Internal validity Internal threats are factors influencing the outcomes of the performed experiment. One potential internal threat is that we focus on automatically detectable smells and this could limit the applicability of the approach since semantic-driven smells might not be representable with the Edelta DSL syntax. Moreover the correctness of the experiments results are driven by the solver, the applied refactoring, smells definitions and quality characteristics calculation formulas. All these elements are defined by modelers and then subject to possible inconsistencies that could influence the final result. To mitigate this aspect we reused, when possible, existing definitions from literature and represented them faithfully with the corresponding models or DSL syntax.

External validity In this context we discuss how the conducted experiment would still be valid outside the used setting. To mitigate this aspect, we considered various models since the dataset is heterogeneous and used in other experiments in literature (Nguyen et al. 2019). We plan to further replicate the experiment with other large datasets.

Throughout the paper we have picked four quality characteristics as a proof of concept to measure the quality of the refactored models. Likewise, we work with a set of eight smells and their corresponding refactorings. Many other characteristics could be measured in the models and other smells could be identified together with different refactorings. We consider the set of characteristics, smells and refactorings representative enough since they are related to different elements in the models, covering a wide range of structural changes in them.

Finally, the examples in the paper are based on EMF and Ecore models, but as we explained, it is possible to switch to other modeling frameworks by extending PARMOREL. Within EMF, the work presented in this paper is specific for Ecore models. However, it could be applied in general to models

instances if the refactoring actions retrieved from the framework were domain specific.

7. Related Work

This section discusses relevant works that are related to smells detection and code refactoring with ML, model refactoring, ML approaches for MDE and, recommender systems.

Smells detection and code refactoring with ML ML for smells detection has been more applied at code level than at model level. In (Fontana et al. 2016), the authors perform a comparative study with different ML techniques for identifying a set of four smells. They achieve high accuracy without needing much data for each smell. However, in the literature review presented in (Azeem et al. 2019) and (Di Nucci et al. 2018), authors point that most studies are done at a theoretical level, and there are still big open challenges the field needs to overcome to reach its full potential.

ML offers the possibility to identify complex smells and it could also be used to detect smells in models. However, users would need to find or define their own datasets in order to tackle the smells they are interested in. The scope of the smells detected using the Edelta DSL is automatically detectable smells, but users just need to define their own smells at code level without needing to train on any dataset. Regarding code refactoring, different ML techniques (Alenezi et al. 2020; Sheneamer 2020) have been applied to predict and identify which parts of the code are prone to be refactored. By doing so, the time spent in refactoring can be reduced. Although our current approach does not support predictions, we use RL to identify both the parts of models that should be refactored in their current state and what is the best action to perform the refactoring. Approaches for code refactoring usually rely on great amounts of data, including code's historic evolution coming from public code repositories. This amount and type of data is not yet available in the MDE field.

Model refactoring The concept of refactoring has been explored using UML class diagrams in (Mens 2006) after a complete analysis of Fowler in (Fowler 1999) for code. A DSL called *Wodel* (Gómez-abajo et al. 2016), allows to create model mutations by means of a metamodel independent specification. Creation, deletion and reference reversal are the primitives offered by the model mutations whereas the composition of mutations are similar to the Edelta mechanism. The specifications are translated into Java code but Edelta works in a different abstraction layer in which the refactoring / mutation is applied.

Similarly to the applied refactorings used in the experiments, a refactoring catalog for UML models is presented in (Sunyé et al. 2001). Whereas in (Xing & Stroulia 2006; Fadhel et al. 2012) mechanisms for detecting refactorings are presented. Lastly, the approach in (Langer et al. 2013) proposes an a searching algorithm for occurrences of composite operations within a set of detected atomic changes in a post-processing manner.

In these approaches, the user is responsible of deciding which refactorings to apply in the model and sometimes of designing them. In PARMOREL, we abstract users from this burden as the tool will take care of deciding which refactorings should be applied to satisfy the user preferences.

ML approaches for MDE We could not find in the literature any research applying RL to model refactoring hence, we focus on other ML techniques as related work.

Puissant et al. propose a tool called Badger based on an artificial intelligence technique called automated planning (Puissant et al. 2015). Badger generates sequences that lead from an initial state to a defined goal.

It has a set of repaired operations to which users can assign costs and weights to decide its priority. Badger generates a set of plans, each plan being a possible way to repair one error. This makes it difficult for the user to decide which action to apply without knowing how it affects the rest of the model. We prefer to generate alternative sequences to refactor the whole model since some actions can modify the model drastically.

It is worth mentioning search-based and genetic algorithm-based approaches since, although they have not been applied yet to model repair, they are possible competitors to RL. These techniques have shown promising results dealing with model transformations and evolution scenarios, for example in (Kessentini et al. 2017) authors use a search-based algorithm for model change detection. These algorithms deal efficiently with large state spaces, however they cannot learn from previous tasks nor improve their performance. While RL is, in the beginning, less efficient in large state spaces, it can compensate with its learning capability. In the beginning, performance might be poor, but with time refactoring becomes straightforward.

Some approaches make use of neural network (NN) architectures to solve different MDE problems. In (Burgueño et al. 2019) authors present a NN architecture for model transformation without specifying code for any specific transformations. Tackling model refactoring, in (Sidhu et al. 2020) authors make use of a deep NN architecture to refactor UML diagrams with symptoms of design flaws. NN need a great amount of data in order to work. The produced solutions are tightly related to the training dataset, so if the requirements of the problem changes, so needs to do the data. By using RL we do not need training data, as these algorithms learn by directly interacting with the models and, by using the abstract concepts of PARMOREL architecture, our tool can easily be adapted to solve different problems without the burden of designing new datasets.

Recommender systems Other approaches such as (Cuadrado et al. 2018; Muşlu et al. 2012) work as recommender systems (both for code and models) instead of only relying on automation. PARMOREL may also be utilized like a recommender system allowing users to choose the solution they prefer from a ranked list of proposed solutions. These choices are in turn fed back to the learning algorithm and affect the rewards (Barriga, Heldal, et al. 2020). However, the main focus of this paper is on providing automatic model refactoring to remove smells. Hence, instead of letting the users know about the consequences of the refactorings so that they decide a solution, we ask them beforehand which consequences (quality characteristics as preferences) they prefer, and we use these preferences to guide the refactoring phase. In (Cuadrado et al. 2018) authors present a

catalogue of quick fixes, knowing which one of them can solve each problem. In our paper, each smell found by the smell finder has a corresponding refactoring, however, we have worked in scenarios where we had a set of available actions and we did not know which one solved each smell. Finally, although some quick-fix approaches (Cuadrado et al. 2018) might be initially faster than PARMOREL, the idea of our approach is that it learns and streamlines its performance the more models it refactors. As could be seen in the evaluation, with a relatively small dataset we already were able to refactor models in which the issues were known by PARMOREL 37% faster on average.

8. Conclusions and future work

In this paper, we present a new PARMOREL extension to support smells detection and selective refactoring. The approach is able to selectively remove smells that has impact on the quality characteristics expressed as preference by the user. To achieve this, we integrate PARMOREL with a tool that allows modelers to identify smells and refactor them with precise refactorings. This extension is based on the integration of tools, e.g., Edelta, and a model-based quality assessment methodology. We demonstrated how we can solve the trade-off between smells and quality characteristics with a dataset used in the literature, consisting of 404 models extracted from GitHub. The results are positive and show that PARMOREL effectively select the best smells to refactor in order to maintain and, even improve, the quality characteristics expressed by the modeler. We outline that this approach is totally model-based and that can be further extended with other preferences, issues and actions that we plan to investigate. The main strength of PARMOREL is the degree of flexibility it provides to the user.

In this flexible environment, we use reinforcement learning to learn how to refactor a model without any prior knowledge of the model, and by using our transfer learning approach with experience sharing, we can forward what the framework learns from previous refactorings. Reinforcement learning might have the weakness to provide a slower solution than other approaches during the first refactorings, however, the idea of our approach is that the learning module learns and streamlines the performance the more models it refactors.

Currently, PARMOREL is limited to quantitative user preferences and it needs to get a set of actions to modify the model, unlike other approaches these actions cannot yet be inferred from the issues in the models. Also, PARMOREL needs to detect issues in a model in order to improve it, it cannot deal with models without issues yet. We plan to address these limitations as part of our future work.

Next, we plan to create a benchmark using different model datasets, including the one used in this paper, with which we will compare PARMOREL results and its performance to other existing model refactor and repair approaches in the literature. Also, we plan to extend PARMOREL to solve other problems relevant in the modeling field, like model refactoring after their corresponding metamodel evolves (co-evolution) and making architectural models compliant with best practices and recommended design patterns.

Additionally, we plan to extend the learning module with other algorithms beyond reinforcement learning, specially focusing in other AI and search-based approaches and study their performance with respect to RL algorithms.

References

Alenezi, M., Akour, M., & Al Qasem, O. (2020). Harnessing deep learning algorithms to predict software refactoring. *Telkomnika*, *18*(6).

Al-Jáafer, J., & Sabri, K. E. (2007). *Metrics for object oriented design (mood) to assess java programs* (Tech. Rep.).

Arendt, T., & Taentzer, G. (2013). A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering*, *20*(2), 141–184.

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, *108*, 115–138.

Babur, O. (2019, March). *A labeled Ecore metamodel dataset for domain clustering.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.2585456

Barriga, A., Heldal, R., Iovino, L., Marthinsen, M., & Rutle, A. (2020). An extensible framework for customizable model repair. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 24–34). ACM.

Barriga, A., Mandow, L., Perez de la Cruz, J. L., Rutle, A., Heldal, R., & Iovino, L. (2020). A comparative study of reinforcement learning techniques to repair models. In *2020 ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C).* ACM.

Barriga, A., Rutle, A., & Heldal, R. (2020, July). Improving model repair through experience sharing. *Journal of Object Technology*, *19*(2), 13:1-21.

Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2016). A customizable approach for the automated quality assessment of modelling artifacts. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 88–93). IEEE.

Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2019). A tool-supported approach for assessing the quality of modeling artifacts. *Journal of Computer Languages*, *51*, 173–192.

Beck, K., & Fowler, M. (2018). Bad smells in code. In *Refactoring: Improving the design of existing code* (2nd ed., chap. 3). Addison-Wesley.

Bellman, R. (2013). *Dynamic programming*. Courier Corporation.

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend* (2nd ed.). Packt Publishing Ltd.

Bettini, L., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2017). Edelta: An approach for defining and applying reusable metamodel refactorings. In *Proc. MODELS (Satellite Events)* (pp. 71–80). ACM.

Bettini, L., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2019). Quality-driven detection and resolution of metamodel smells.

IEEE Access, *7*, 16364–16376.

Bettini, L., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2020). Edelta 2.0: Supporting live metamodel evolutions. In *Proc. MODELS (Satellite Events)* (pp. 1–10). ACM.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on software engineering* (pp. 592–605). IEEE Computer Society Press.

Burgueño, L., Cabot, J., & Gérard, S. (2019). An LSTM-Based Neural Network Architecture for Model Transformations. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 294–299). IEEE.

Cuadrado, J. S., Guerra, E., & de Lara, J. (2018). Quick fixing atl transformations with speculative analysis. *Software & Systems Modeling*, *17*(3), 779–813.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018). Detecting code smells using machine learning techniques: are we there yet? In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)* (pp. 612–621). IEEE.

Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2014). Mining metrics for understanding metamodel characteristics. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering* (pp. 55–60). ACM.

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on software engineering*, *21*(2), 146–162.

Fadhel, A. B., Kessentini, M., Langer, P., & Wimmer, M. (2012). Search-based detection of high-level model changes. In *Icsm* (pp. 212–221). IEEE Computer Society.

Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*(3), 1143–1191.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley.

García-Magariño, I., Gómez-Sanz, J. J., & Fuentes-Fernández, R. (2008). An Evaluation Framework for MAS Modeling Languages Based on Metamodel Metrics. In *AOSE* (Vol. 5386, pp. 101–115). Springer.

Genero, M., & Piattini, M. (2001). Empirical validation of measures for class diagram structural complexity through controlled experiments. In *QAOOSE@ECOOP*.

Gómez-abajo, P., Guerra, E., Lara, J. D., Gomeza, P., & Guerra, E. (2016). Wodel : A Domain-Specific Language for Model Mutation. In *SAC* (pp. 1–6). ACM.

Iovino, L., Barriga, A., Rutle, A., & Heldal, R. (2020). Model repair with quality-based reinforcement learning. *Journal of Object Technology*, *19*(2), 17.

Kessentini, M., Mansoor, U., Wimmer, M., Ouni, A., & Deb, K. (2017). Search-based detection of model level changes. *Empirical Software Engineering*, *22*(2), 670–715.

Kolovos, D. S., Paige, R. F., & Polack, F. A. (2006). The epsilon object language (EOL). In *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 128–142). Springer.

Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., & Kappel, G. (2013). A posteriori operation detection in evolving software models. *J. Syst. Softw*, *86*(2), 551–566.

López-Fernández, J. J., Guerra, E., & De Lara, J. (2014). Assessing the Quality of Meta-models. In *MoDeVVa@MODELS* (pp. 3–12). CEUR-WS.org.

Mens, T. (2006). On the use of graph transformations for model refactoring. In *GTTSE (Revised Papers)* (pp. 219–257). Springer.

Mumtaz, H., Alshayeb, M., Mahmood, S., & Niazi, M. (2019). A survey on UML model smells detection techniques for software refactoring. *Journal of Software: Evolution and Process*, *31*(3).

Muşlu, K., Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. (2012). Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices*, *47*(10), 669–682.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., & Iovino, L. (2019). Automated Classification of Metamodel Repositories: A Machine Learning Approach. In *MODELS* (pp. 272–282). IEEE.

Ortega, M., Pérez, M., & Rojas, T. (2003). Construction of a systemic quality model for evaluating a software product. *Software Quality Journal*, *11*(3), 219–242.

Puissant, J. P., Van Der Straeten, R., & Mens, T. (2015). Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, *14*(1), 461–481.

Sheldon, F. T., & Chung, H. (2006). Measuring the complexity of class diagrams in reverse engineering. *Journal of Software Maintenance*, *18*(5), 333–350.

Sheneamer, A. M. (2020). An Automatic Advisor for Refactoring Software Clones Based on Machine Learning. *IEEE Access*, *8*, 124978–124988.

Sidhu, B. K., Singh, K., & Sharma, N. (2020). A machine learning approach to software model refactoring. *International Journal of Computers and Applications*, 1–12.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework* (2nd ed.). Addison-Wesley.

Strittmatter, M., Hinkel, G., Langhammer, M., Jung, R., & Heinrich, R. (2016). Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. In *CEUR Workshop Proceedings* (Vol. 1706, p. 30-39). CEUR.

Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.-M. (2001). Refactoring UML Models. In *Proceedings of UML* (Vol. 2185, pp. 134–148). Springer.

Thrun, S., & Littman, M. L. (2000). Reinforcement learning: an introduction. *AI Magazine*, *21*(1), 103–103.

Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE software*, *31*(3), 79–85.

Xing, Z., & Stroulia, E. (2006). Refactoring Detection based on UMLDiff Change-Facts Queries. In *WCRE* (pp. 263–274). IEEE.

About the authors

Angela Barriga is a PhD Candidate at Western Norway University of Applied Sciences. She has experience working with machine learning, computer vision, gerontechnology and pervasive systems. Barriga's thesis is focused on model repair, specially on repairing using reinforcement learning. She has been part of the local organization of iFM 2019 and is involved in STAF 2020-2021. She is also part of the program committee of the third international workshop on gerontechnology. You can learn more about her at https://angelabr.github.io/ or contact her at abar@hvl.no.

Lorenzo Bettini is an Associate Professor in Computer Science at DISIA Dipartimento di Statistica, Informatica, Applicazioni 'Giuseppe Parenti', Università di Firenze, Italy, since February 2016. Previously, he was an Assistant Professor (Researcher) in Computer Science at Dipartimento di Informatica, Università di Torino, Italy. His research interests cover design, theory and implementation of DSLs and programming languages (in particular Object-Oriented languages and Network aware languages). He is also committer of the Eclipse projects Xtext and SWTBot and the project lead of the Eclipse project EMF Parsley. Contact him at lorenzo.bettini@unifi.it, or visit http://www.lorenzobettini.it.

Ludovico Iovino is Assistant Professor at the GSSI Gran Sasso Science Institute, LAquila - in the Computer Science department. His interests include Model Driven Engineering (MDE), Model Transformations, Metamodel Evolution, code generation and software quality evaluation. Currently he is working on model-based artifacts and issues related to the metamodel evolution problem. He has been included in program commitees of numerous conferences and in the local organisation of the STAF 2015 and iCities 2018 conferences, he organised also the models and evolution workshop at MODELS 2018. He is part of different academic projects related to Model Repositories, model migration tools and Eclipse Plugins. Contact him at ludovico.iovino@gssi.it, or visit http://www.ludovicoiovino.com.

Adrian Rutle is professor at Western Norway University of Applied Sciences. Adrian holds PhD in Computer Science from the University of Bergen, Norway. Rutle is professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen. Rutles main interest is applying theoretical results from the field of model-driven software engineering to practical domains and has expertise in the development of modeling frameworks and domain-specific modeling languages. He also conducts research in the fields of modeling and simulation for robotics, eHealth, digital fabrication, smart systems and machine learning. Contact him at adrian.rutle@hvl.no

Rogardt Heldal is professor of Software Engineering at the Western Norway University of Applied Sciences. Heldal holds an honours degree in Computer Science from Glasgow University, Scotland and a PhD in Computer Science from Chalmers University of Technology, Sweden. His research interests include requirements engineering, software processes, software modeling, software architecture, cyber-physical systems, machine learning, and empirical research. Many of his research projects are performed in collaboration with industry. Contact him at rogardt.heldal@hvl.no

APPENDIX

In this appendix, we show how we extend PARMOREL to identify and restore inter-model consistency between UML class and sequence diagrams. These new extensions are part of paper [12], which is in the second round of the review process in the SoSyM journal at the moment of writing this thesis (see Section 1.3).

We extend the issues and actions submodule to find and restore inter-model inconsistencies. The preferences module in this extension uses a coupling calculation technique in order to reward lower coupling in the sequence diagrams when restoring consistency. Next, we present these extensions and test them through a preliminary evaluation.

Issues submodule: Inter-model inconsistencies

To demonstrate that PARMOREL is able to restore inter-model consistency, we have implemented, as an example, the following rules (inspired by rules 110 and 114 from [92]) to identify inconsistencies between UML sequence and class diagrams:

- **Rule 1:** If a message in a sequence diagram refers to an operation through the signature of the message, then that operation must belong, as per the class diagram, to the class that types the target lifeline of the message.
- **Rule 2:** Each public operation in a class diagram triggers a message in at least one sequence diagram.

As an example, we show in Fig. 1 a modification of the video-on-demand system (VoD) presented in [40, 62, 74]. In this example, we have a class diagram with three classes: *Video*, *Server*, and *User*, and a sequence diagram with the lifelines corresponding to these classes. We assume the class diagram has evolved and its corresponding sequence diagram is no longer consistent with the new changes. As can be seen, the operation *disconnect* should be invoked in *Server* by *Video*, however, in the sequence diagram it is invoked in *Video* by *User*, which violates Rule 1. Additionally, the operation *loop* does not appear in any of the classes' lifelines in the sequence diagram, hence violating Rule 2. This operation should be invoked in *Video* either by *User* or *Server*.

With this extension, we not only include a new type of issue, but also we prove that PARMOREL can work with UML models.

Actions submodule: Repairs for inter-model inconsistencies

To solve the inconsistencies presented in the diagram in Fig. 1, we have implemented two actions into the actions submodule. These actions produce the diagram depicted in Fig. 2. As an example, we have implemented actions that modify the sequence diagram, assuming the changes done in the corresponding class diagram have not been propagated yet.

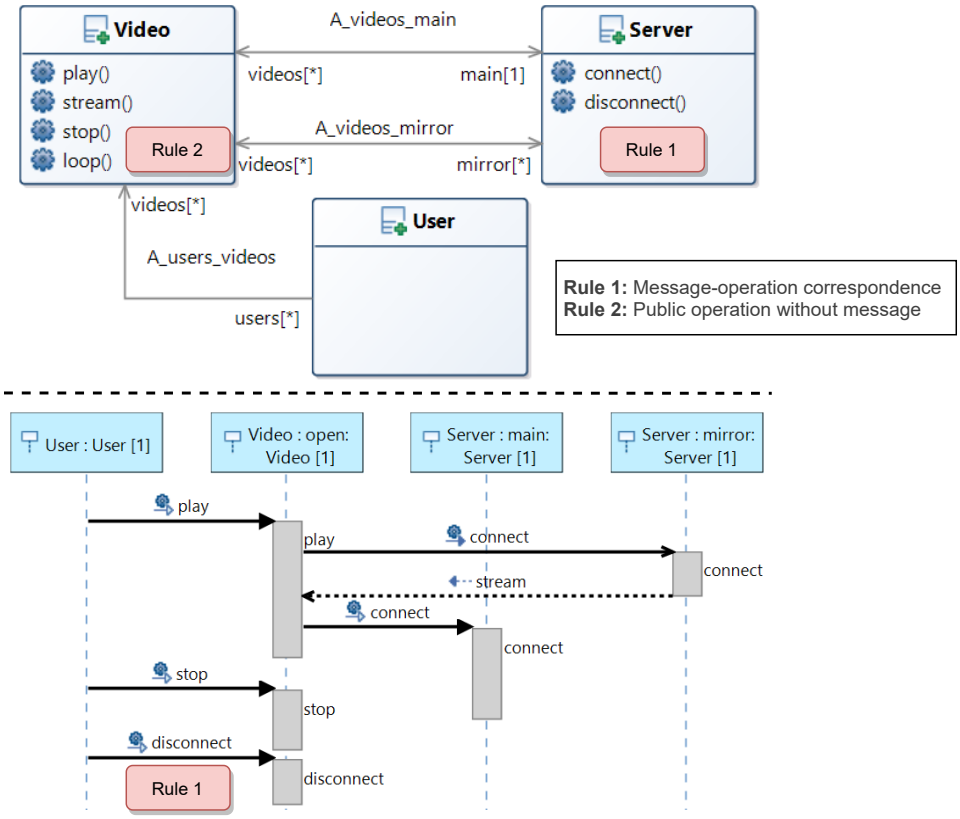


Fig. 1: Example showing inconsistencies between UML class and sequence diagrams violating rules 1 and 2

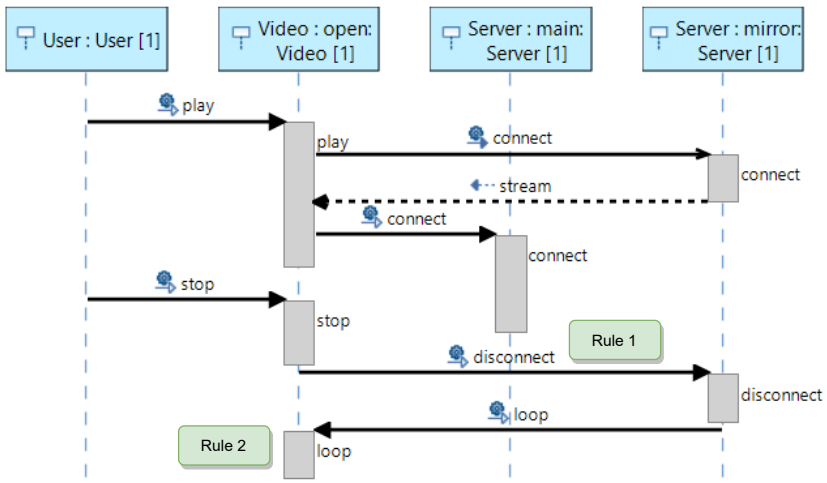


Fig. 2: PARMOREL's solution for the example in Fig. 1

- **Move message:** moves a message between lifelines to its corresponding place according to the class diagram. It solves inconsistencies caused by violating Rule 1.

- **Add message:** adds a message in a lifeline according to its corresponding operation and the class it belongs to in the class diagram. It solves inconsistencies caused by violating Rule 2.

Preference module: Coupling

By using the metrics offered in SDMetrics [94], we can define preferences to guide the repair of UML models. To this end, we sum up the values of *MsgSent* and *MsgRecv* of each lifeline (being n the total number of lifelines, see Equation 1). Then, we divide the sum value by the addition of the *MsgSent* and *MsgRecv* of each lifeline. Finally, we add the obtained value for each lifeline into $reward$ (see Equation 2). This value will be higher as the total coupling of the diagram decreases. We use this value as a reward for the learning algorithm.

$$sum = \sum_{i=1}^{n} MsgSent_i + MsgRecv_i \tag{1}$$

$$reward = \sum_{i=1}^{n} \frac{sum}{MsgSent_i + MsgRecv_i} \tag{2}$$

For example, returning to the example in Fig. 1, the operation *disconnect* should be invoked in *Server* by *Video*, but there are two possible *Server* lifelines: *mirror* and *main*, each leading to a different coupling value. Likewise, the operation *loop* should be invoked in *Video* either by *User* or one of the *Server* lifelines. For this example, the optimal solution found by PARMOREL (see Fig. 2) has a reward of 26.5. Here, PARMOREL moves the operation *disconnect* to have the *Server:mirror* lifeline as receiver and *Video* as sender and it adds the operation *loop* to have the *Video* lifeline as receiver and *Server:mirror* as sender. Other solutions receive a lower reward, since their coupling is worse, for example, moving the operation *disconnect* and adding *loop* to have, in both cases, the *Server:main* lifeline as the receiver and *Video* as the sender would get a reward of 20.66. In this case, both *Video* and *Server:main* lifelines would have worse coupling.

Evaluation: Restoring inter-model consistency while lowering coupling

To perform a preliminary initial evaluation, as a dataset, we have manually created 12 models, 6 pairs of class and sequence diagrams using the Eclipse IDE of UMLDesigner 9.0 [21]. The sequence diagrams are based on sequence diagrams that can be found in [38]. Regarding the class diagrams, we created them based on these sequence diagrams, since in [38] they were not available. Furthermore, we arbitrarily added the inter-model inconsistencies in the sequence diagrams, as the diagrams available did not contain

this kind of issues. The subject sequence diagrams have between 4 and 11 lifelines and include between 2 and 10 violations of rules 1 and 2.

In this experiment, we evaluate whether PARMOREL is able to deal with unidirectional inter-model inconsistencies between UML class and sequence diagrams. We want to evaluate that the framework can be extended to deal with different types of models and issues.

For each pair of class and sequence diagrams, first, we analyze if there exist any violations of rules 1 and 2. Then, for every violation detected, PARMOREL obtains which potential senders and receivers the repair actions could have. Then, in every episode, PARMOREL applies the repair actions with different senders and receivers, obtaining the coupling of the repaired sequence diagrams. Finally, the sequence of repair actions (with the best combination of senders and receivers) that leads to the lowest coupling is selected. For each pair of diagrams, it takes PARMOREL between 0.7 and 8.2s to learn how to restore the consistency.

PARMOREL is able to restore all consistencies from the dataset models, always choosing the most optimal solution with respect to coupling (the sequence of actions that create the model with the lowest coupling), hence providing personalized restoration. With these results, we can conclude that PARMOREL can support different types of models (UML class and sequence diagrams) and more complex issues, like inter-model inconsistencies.

Additionally, while performing this evaluation, we discovered that PARMOREL is able to design sequence diagrams from scratch. For every operation existing in a class diagram, a violation of Rule 2 is triggered (*each public operation in a class diagram triggers a message in at least one sequence diagram*), since no messages are existing in an empty sequence diagram. Then, PARMOREL is able to create a sequence diagram with the most optimal distribution of messages between the lifelines in the sequence diagram to reduce the coupling and create a diagram with such a distribution. Hence, apart from repairing, our approach could be used to assist modelers—with some degree of auto-completion—when designing inter-related models. This design could be guided by a reference model and different user preferences.

We find this discovery could have great potential to automatically generate models. Apart from keeping consistency between corresponding models, PARMOREL could help modelers to design new models in inter-modeling environments. This design could be guided by different user preferences, like reducing coupling, enhancing cohesion, improving quality characteristics, etc. We plan to continue researching the potential of this discovery in the future.