Software and Systems Modeling (2022) 21:1739–1762
https://doi.org/10.1007/s10270-022-01005-0

SPECIAL SECTION PAPER

PARMOREL: a framework for customizable model repair

Angela Barriga[1] · Rogardt Heldal[1] · Adrian Rutle[1] · Ludovico Iovino[2]

Received: 1 March 2021 / Revised: 11 February 2022 / Accepted: 21 March 2022 / Published online: 4 May 2022
© The Author(s) 2022

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. 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, these tools are often inflexible for customization and hard to extend. This paper presents a customizable and extensible model repair framework, PARMOREL, that enables users to deal with different issues in different types of models. The framework uses reinforcement learning to automatically find the best sequence of actions for repairing a broken model according to user preferences. As proof of concept, we repair syntactic errors in class diagrams taking into account a model distance metric and quality characteristics. In addition, we restore inter-model consistency between UML class and sequence diagrams while improving the coupling qualities of the sequence diagrams. Furthermore, we evaluate the approach on a large publicly available dataset and a set of real-world inspired models to show that PARMOREL can decide and pick the best solution to solve the issues present in the models to satisfy user preferences.

Keywords Model repair · Reinforcement learning · Customizable framework

1 Introduction

In model-driven software engineering, the increasing complexity of software systems is handled by utilizing abstract software models. When producing these models, software developers may introduce various issues which corrupt the models or reduce model quality, such as syntactic errors, smells [1], antipatterns [2], inter-model inconsistencies [3]. In addition, the chances of corrupting a model increase along with the size of development teams and the number of software requirements [4], lack of coordination, misunderstanding, and mishandled collaborative projects. These issues may lead to severe challenges in the later phases of the development process since the reliability and accuracy of these models are important to correctly produce the software systems which they represent.

Ensuring that a model does not contain issues is a time-consuming task. As a consequence, a variety of approaches to automatic model repair have been proposed over the last decades to tackle the repair of corrupted models from different perspectives and applied to different models [5–7]. Despite the existence of automatic approaches, there might be multiple repair solutions which do not satisfy all modelers preferences. Consequently, the modeling community has developed a series of metrics and characteristics to get an unbiased measure of a model's quality, for example: analyzability, adaptability, and understandability [8,9]. Even though quality characteristics have been extensively studied in the literature [10–13], the quality of the automatically repaired models has not been the main focus of existing repairing tools.

In the same direction, model distance is another factor which could be considered during repair to objectively maintain a model's quality. The literature has highlighted the importance of preserving the original model structure while

Communicated by S. Abrahão, E. Syriani, H. Sahraoui, and J. de Lara.

✉ Angela Barriga
abar@hvl.no

Rogardt Heldal
rohe@hvl.no

Adrian Rutle
aru@hvl.no

Ludovico Iovino
ludovico.iovino@gssi.it

[1] Western Norway University of Applied Sciences, Bergen, Norway

[2] Gran Sasso Science Institute, L'Aquila, Italy

repairing, in order to minimize undesired side-effects in the repaired model [4,14]. A tool to measure the preservation of the original model structure is the calculation of the distance between the original and the repaired model [15–17]. Hence, integrating model distance calculation in repairing tools could produce repaired models of high quality.

Furthermore, software models are diverse since they represent different aspects of software systems, Accordingly, repair tools are required to handle different model types (e.g., UML class and sequence diagrams, Ecore class diagrams, etc.). However, most tools are tailored to one type of model and, if this type changes, new tools need to be developed to repair the models. Although there exist approaches to repair different types of models (Ecore, different UML models), with different types of issues (syntactic and semantic errors, smells, etc.), and to measure the quality of the produced models (quality characteristics, model distance, etc.), to the best of our knowledge, there is no tool that integrates all these different aspects of model repair. This paper's main goal is to fill this gap with an extensible framework.

In previous work [18–22], we presented our approach to model repair using our PARMOREL framework, which finds a sequence of repair actions according to preferences introduced by the user. PARMOREL stands for Personalized and Automatic Repair of software MOdels using and REinforcement Learning (RL) algorithms [23]. PARMOREL provides structural model repair, it repairs models representing a structure, but not a behavior whose distance and repair plan would require to consider execution semantics. To simplify, throughout the paper we will refer to structural model repair as model repair. So far, PARMOREL has been evaluated for repairing syntactic errors in Ecore class diagrams [18], selective removal of model smells [22], reusing learning to streamline new repairs [19], and repairing syntactic errors while improving quality characteristics [20]. In [21], we presented the extensible potential of PARMOREL preferences and repaired syntactic errors in Ecore class diagrams while reducing the model distance.

In this paper, we give, for the first time, a complete overview of the PARMOREL framework, including a detailed explanation of PARMOREL's modular architecture. Here, we extend [21] by adding new extensions for issues, actions, and preferences to deal with inter-model inconsistencies in UML models. The new extensions allow PARMOREL to: (1) support UML models (class and sequence diagrams), (2) restore inter-model consistency, and (3) reduce coupling as a user preference. Unlike our previous works, the focus of this paper is not to show that PARMOREL can solve a specific set of issues under some set of preferences, but to detail its modular architecture and demonstrate the extensibility of its modules through a series of implementations.

We demonstrate the extensibility potential of PARMOREL by allowing users to (1) choose, add and modify repair preferences, (2) work with different types of models, (3) edit which issues are detected, and (4) with which actions they are addressed, as well as (5) customize the learning algorithm of the framework. Furthermore, we evaluate the extensibility of PARMOREL through two different experiments: (1) repairing syntactic errors in Ecore class diagrams while preserving the original model structure [21] and (2) restoring inter-model consistency in UML class and sequence diagrams while reducing the coupling in the sequence diagrams.

To the best of our knowledge, there are no other approaches to model repair which utilize reinforcement learning and allow for personalization of repair objectives. Instead of having a specific tool for each kind of issue and each type of model, PARMOREL works as a unified extensible framework, which can be extended to support the new issues that modelers need to solve in their models. With this in mind, this paper contributes to the model repair field by applying reinforcement learning in a framework which allows for personalization and extension.

1.1 Structure of the paper

This paper is organized as follows: Sect. 2 presents a motivating example and an analysis of corrupted models taken from GitHub repositories. Section 3 introduces background about how PARMOREL uses RL to repair models. Section 4 introduces and explains our approach, presenting PARMOREL's modules and the implementations we have created for them. Sections 5 and 6 present experiments where we extend and apply PARMOREL to address syntactic errors and inter-model inconsistencies, respectively. Then, we present threats to validity in Sect. 7, explore the related work in Sect. 8 and conclude the paper in Sect. 9.

2 Motivation

In this section, we will use a domain model as a motivating example of why model repair is essential. A domain model is often represented as a class diagram, showing concepts of the domain and its relationships. In particular, we have retrieved Ecore class diagrams from GitHub repositories, which are collected in a dataset used in [24,25]. The dataset contains a total of 555 class diagrams. The diagrams contain between 8–213 classes, and between 30–186 features, including attributes, references and operations. This dataset is available to download in [24].

We have filtered the dataset and extracted the models which are corrupted with syntactic errors and model smells. In the first filtering, we discovered that 107 models (almost 20% of the dataset) contained a set of syntactic errors with a total of 973 error occurrences. We present details about the errors found in the dataset in Table 1, where we report the

error code and associated error description. These errors are retrieved by the EMF [26] platform that checks validation errors.

In the second filtering of the dataset, we analyzed whether the models contained smells. Smells usually represent a symptom of poor decisions during the models' design phase [27]. Smells in modeling [1] are indicators that something may be wrong within the model design, even if the model is valid. In our analysis, we took into account the following smells, which are part of an automatic detection mechanism presented in [1]:

1. Concrete abstract class
2. Duplicated features in hierarchy
3. Duplicated features in classes
4. Dead class
5. Redundant container relation
6. Abstract subclasses of concrete superclass
7. Abstract concrete class
8. Classification by hierarchy

Some of these smells might seem similar to the errors presented in Table 1, for example, the smells regarding duplicated features and E7. In this case, the smells refer to features (references, attributes or operations) that are duplicated in different classes or in a hierarchy, while E7 indicates a duplication inside the same class. Furthermore, on the existence of smells models are valid and compile, while when there are errors, they are retrieved as the causes of a failed compilation. Only 58 models from the total 555 in the dataset were smell-free, meaning that 497 models (89.54% of the dataset) present some type of smell.

As an example, in Fig. 1 we show an occurrence of E7, where two features with the same name occur in a class. To be precise, the class WebApp inherits the features of NameElement and for this reason, the feature name appears to be declared twice. Moreover, the class Service seems to have an unresolved proxy as supertype (E13). An unresolved proxy is a reference to an external element contained in another domain model, usually imported or cross-referenced. This error is highlighted in Fig. 1 with "–> null," referring to an unresolved superclass of Service.

It is worth noticing that this model can be "repaired" in different ways. For instance, error E7 can be repaired by removing the class NameElement or by removing the attribute name in the WebApp class, or by removing it from NameElement. E13 could be repaired, for example, by removing the faulty reference, correcting it or removing the referenced class.

Concerning the smells, the class WebApp is declared as abstract, whereas the supertype is concrete. This smell is called *Abstract concrete class* and should be resolved by inverting the abstract property of the two indicted classes, or by making the NameElement class abstract, or making both classes concrete.

Refactoring this model in different ways could affect the quality characteristics of the model, for instance, the understandability of the model could be affected if we remove the hierarchy. In addition, if we remove a class to make the model valid again, the size and structural composition of the model would change making the resulting repaired model quite different from the initial version. Moreover, removing some smells might affect negatively the overall quality of the model and it might be better to ignore them.

We find the number of models containing errors and smells in this dataset an indicator for the need to support model repair as an automated activity. The fact that the errors and smells are not straight forward to solve, meaning they might

Table 1 Occurrences of syntactic 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 classifiers with the same name	2
E7	Two or more features 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
E13	Unresolved proxy	90

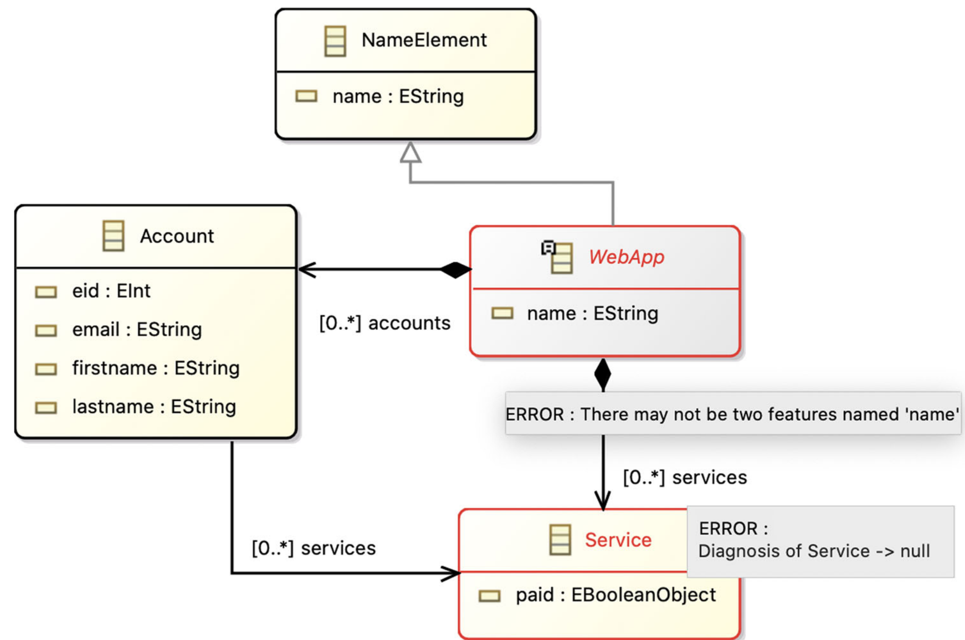


Fig. 1 Motivating example of a class diagram containing syntactic errors and a smell

have different potential solutions, is also an indicator for the need to support modelers with extensible repair preferences so that they can customize which kind of repair actions satisfy their specific needs (quality characteristics, model distance, specific project-related metrics, etc.).

Likewise, apart from syntactic errors and smells, models might present other issues like semantic problems, inter-model inconsistencies, etc. Implementing automated tools for repairing and improving these models is time and resources consuming, whereas it involves complex optimization problems. Instead of having a specific tool for each kind of problem and for each type of model, we propose in this paper a unified extensible framework, PARMOREL, which could be extended to support the new issues that modelers need to solve in their models.

3 Background

In this section, we provide background about how PARMOREL uses RL to repair models. In the following, we present the theory we have developed in order to adapt RL concepts to solve the model repair problem.

We chose RL to solve the model repair problem due to the lack of data available in the modeling field, which makes it difficult to apply most ML algorithms. 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 [28–32].

RL algorithms are a solution that allows personalization of results without needing large amounts or pre-labeled data.

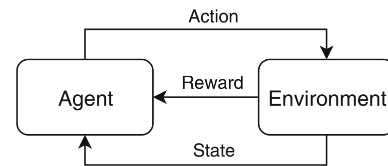


Fig. 2 Agent-environment interaction in RL

Due to RL's 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 constitute the model repair problem. Also, several mechanisms are supported so that users can interact and provide their feedback to these algorithms.

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 rewards for each of these actions [23]. 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 2 displays this interaction.

For example, if the *environment* is a maze, the *agent* is a robot, the *action* is walking one step to the right and the *state* the current position of the robot in the maze, then, the new state would be the robot's new location: one position to the right. If the action is positive for the agent (moving toward the maze exit), 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., entering the exit of the maze.

RL concepts can be used to solve different problems. Each problem might require a different definition of these concepts to be solved (e.g., a state could be a position in a maze, the score in a videogame, etc.). In the following, we detail some RL concepts important to understand the rest of the paper together with our definitions for solving the model repair problem:

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. In the model repair problem, we define the state space by the set of issues present in the model. In [21] we introduced the concept of *issue*. An *issue* represents something that is improvable in a model. An issue could be a syntactic error, a smell, a violation with respect to an architectural pattern or a specific constraint, an inconsistency between two or more models, etc. 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: {*issue1, issue2, issue3*}.
2. Action space: The set of actions that can modify the system, leading to new states. In the model repair problem it is the set of editing actions able to repair a model. Actions might come from an external tool, be defined by users or obtained from a modeling framework such as EMF [26], as we will see in the next section. 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 syntactic issues, are: *delete, setName, setType, setContainment*, etc.
3. Step: A step corresponds to the application of one action in the system. In the model repair problem, a step corresponds with the application of one action to solve an issue in the model.
4. 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 straight forward to conclude what number is the right one in a given context [23] and hence, needs to be defined empirically through experimentation [33]. In the model repair problem, 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.
5. Reward: A numerical value that tells the agent how good is the action it applies. When repairing models, in every non-final step (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 metrics 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 optimizing quality metrics). Users can choose their preferences before the repair process starts.

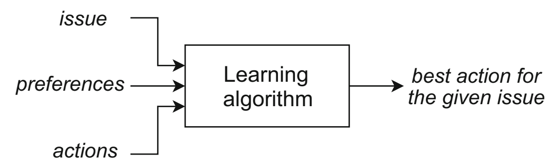


Fig. 3 Simplified workflow in PARMOREL

4 PARMOREL framework

In this section, we present our proposed approach to model repair along with its implementation in the PARMOREL framework. PARMOREL uses 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. 3). The framework also contains a learning algorithm, specifically RL, in charge of learning and deciding which is the best action—among a set of available actions—to address an issue, according to the user's preferences.

The PARMOREL framework permits the issues, actions, preferences, and learning 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. 4), and permits users to customize its modules through a series of interfaces. In this section, we explain our RL implementation and each of the constituting modules of PARMOREL.

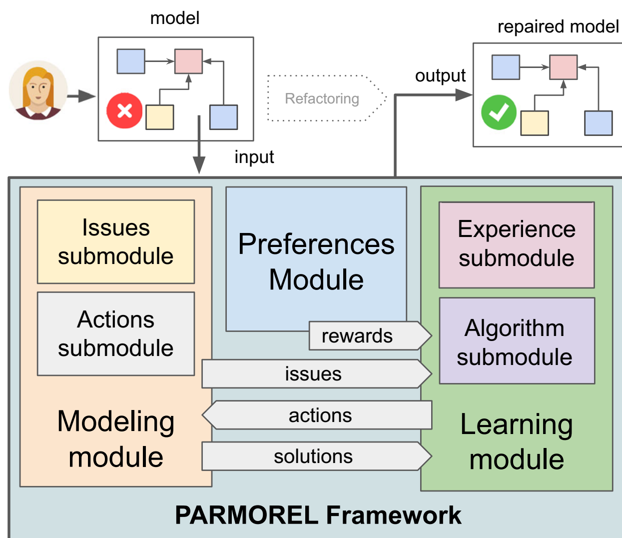


Fig. 4 PARMOREL's modular architecture

4.1 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 action should address each issue.

When adding new types of models in PARMOREL, the main effort is in extending the *modeling module* with a mechanism to identify issues in the models and to solve them (in this section, the Eclipse Modeling Framework (EMF), Edelta and SDMetrics to solve syntactic errors, smells and inter-model inconsistencies). PARMOREL is detached from the model type due to the abstract concepts—issues, actions, preferences—on which it is founded.

The *modeling module* is divided into two submodules, namely the *issues submodule* and *actions submodule*, as shown in Fig. 4. Next, we present the implementations we have developed for each submodule. For these implementations, we have used the APIs and DSLs of external tools, as presented in the following.

4.1.1 Issues submodule

The *issues submodule* is in charge of identifying which issues are present in the model and send them to the *learning module*. For example, we have developed the following implementations of the *issues submodule*: syntactic errors, smells and inter-model inconsistencies. Next, we present each of these implementations. A detailed list of the issues implemented so far can be found in [34].

Syntactic errors We have used the EMF diagnostician, to implement the identification of syntactic errors [21] that violate certain constraints of the Ecore metamodeling language [26] 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.). With this implementation, the approach is able to repair issues that belong to the catalogue of errors provided by the EMF diagnostician.

Smells By using EMF [26] together with Edelta [35], 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 DSL, for easily defining Ecore model evolutions and refactorings. The core features of Edelta and its DSL have been detailed in [35]. By using the Edelta DSL, users can provide the specification of custom smell finders, which can be properly organized in reusable libraries. A smell finder is associated with a refactoring for smell removal.

Inter-model inconsistencies In this extension, we have used SDMetrics [36] 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. In our context, we use various UML models used to describe different aspects of a software. For instance, class diagrams to describe the structure and sequence diagrams to define (parts of) the behavior of the software. These models should be kept consistent with each other since inconsistencies between models may be a source of faults during software development activities that rely on these models [37].

4.1.2 Actions submodule

The *actions submodule* is in charge of:

- Sending to the *learning module* which actions are available for modifying the model.
- 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 always defined as an answer to the issues present in the model. Hence, for every implementation of the *issues submodule*, there is a corresponding *actions submodule* implementation. For example, when dealing with syntactic errors (e.g., two classifiers with the same name), we will need to provide PARMOREL with a set of editing actions that could solve the errors (e.g., delete or rename).

This submodule is designed so that actions cannot create a loop. A loop could happen when repairing an issue with a certain action leads to introducing another issue, which, when

repairing it may lead back to the first issue, hence getting stuck in a never-ending cycle. As presented in Sect. 3, an action applied to solve an issue is considered a step, and PARMOREL counts with a maximum number of steps that can be applied per episode. By limiting the number of steps, if by the end of the episode PARMOREL has not been able to find a repair solution due to getting stuck in a loop, the actions responsible for the loop will receive a negative reward in order to not being chosen in the future.

So far, we have developed the following implementations of this submodule: repairs for syntactic errors, refactorings for smells, and repairs for inconsistencies. Next, we present each of these implementations.

Repairs for syntactic errors In order to repair syntactic errors, we make use of the actions available within EMF to modify Ecore class diagrams [38]. 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 implementation are: setName(), setOppositeReference(), removeSuperType(), etc. In this implementation, we take all actions available in the EMF and filter them, keeping only those that can be invoked to modify the model parts containing issues (e.g., setName or remove in a classifier).

Refactorings for smells In this implementation, we use the Edelta DSL together with EMF to specify model refactorings [22]. 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 syntactic errors, 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.

Repairs for inconsistencies In this implementation, we use the Java DOM libraries to manipulate the UML models' structure as XML files. We have implemented operations that allow us to create new messages in the sequence diagram and move existing ones.

Unlike the previous issues, most inconsistencies have a single action to restore them, but the actions might be applicable in 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.

This is relevant because in other repair scenarios an action could only be applied in a single way to repair an issue (e.g., renaming an attribute). When dealing with issues with respect to messages in a sequence diagram, however, PARMOREL has to identify the potential senders and receivers of the repair actions, and finding what is the best combination according to the preferences selected by the user. Hence, the tool focuses on finding the best option regarding where to apply an action rather than finding the action which should be used to perform the repair.

4.2 Preferences module

Users can customize the results PARMOREL produces with their own preferences by implementing the *preferences module* (see Fig. 4). A preference indicates which kind of actions the user wants to be applied 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. A preference could be the prioritization of quality characteristics, boosting specific measurements of the model, etc. For example, if a user wants to produce models which are better with respect to a particular quality characteristic, PARMOREL would choose actions which 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). Users can create their own preferences by implementing PARMOREL modules through a series of interfaces [34]. When adding new preferences users have to indicate the value that corresponds with each preference, either a static value or a calculation function (as we will see with the implementations presented in this section). The preferences are not limited to a certain level of expressivity (type or instance model level, syntactic or semantic, etc.). As long as a preference provides a numeric value that can be used as a reward, it will be supported by PARMOREL.

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 preferences. More details about how rewards work will be provided in the following section.

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

So far, we have extended this module implementing the following preferences: quality characteristics, model distance, and coupling. Next, we present each implementation.

Quality characteristics In this implementation [38], we use quality characteristics extracted from the literature [10–12] as user preferences. PARMOREL integrates a quality evaluation tool which is inspired by [39].

This quality evaluation tool not only provides an evaluation mechanism but also supports the specification of custom quality characteristics. So far, we have specified the following quality characteristics to be used as user preferences: maintainability, understandability, complexity, and reusability. For more details about these quality characteristics, we refer the reader to our previous work [38]. By using this implementation, PARMOREL can learn how to repair Ecore class diagrams in a way that the selected quality characteristics are improved. The values of the quality characteristics will be used as rewards and hence PARMOREL will be able to choose the repair actions that lead to models with better quality.

Model distance In [21], we exemplify an implementation of the *preferences module* by using a model distance metric to guide the repair of Ecore class diagrams. The concept of model distance has been previously used in the literature especially related to model comparison [40]. The conceptual distance between two models can be obtained by comparing them and processing commonly occurring concepts by also counting the elements that only exist in one of the models [41]. 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 an Epsilon Comparison Language [42] (ECL) script, which can be customized.

Coupling By using the metrics offered in SDMetrics [36], 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) [43] 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, so 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 loose as possible.

4.3 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 to streamline following repairs. The *learning module* is divided into two submodules, namely the *algorithm submodule* and the *experience submodule*.

4.3.1 Algorithm submodule

This submodule contains our implementation of how to apply RL to repair models. We have used tabular RL algorithms, such as Q-learning and Q(λ) [23]. 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. 5. 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 with which it 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 [33]). 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 ith 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 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.

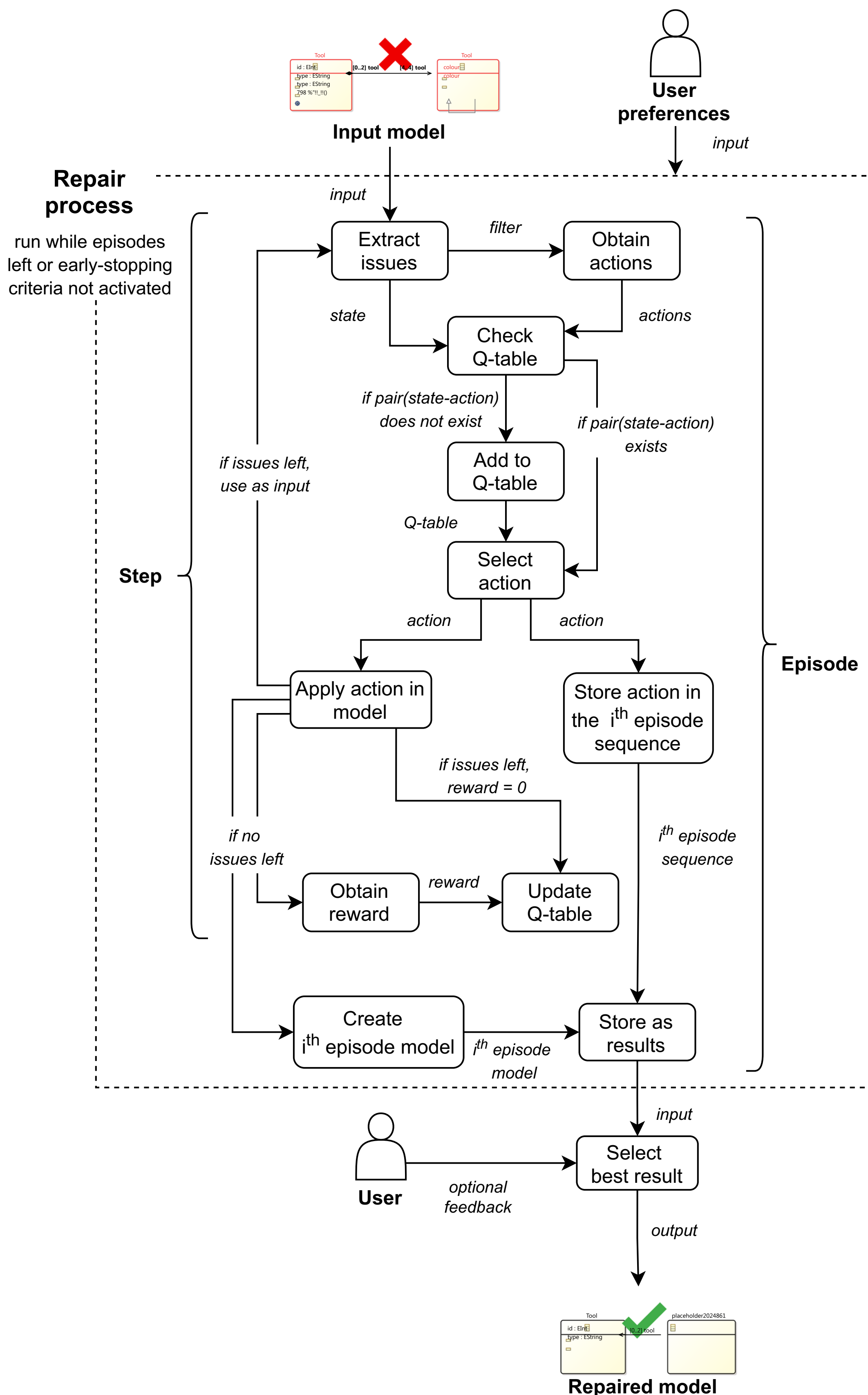


Fig. 5 Summary of the repair process using RL

However, if there are no issues left in the model, following the arrow *if no issues left*, the action Obtain reward is triggered, here, 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 ith episode model creates a repaired version of the model with all the actions applied during the episode. Following the arrows ith *episode model* and ith *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 get 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 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 [33]).

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 highest Q-values will be selected. Lastly, following the arrow *output*, the corresponding provisional repaired model is saved as the final 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, the 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.

In previous work [33], we have implemented this submodule with the following algorithms: Q-Learning, Q(λ), Monte Carlo, SARSA and, SARSA(λ). We compared the performance of these different RL algorithms in PARMOREL and Q(λ) was the one that provided us the best performance. Hence, it is the one we use in our current implementation and the one we describe in this section. Since the main topic of this paper is not about RL algorithms, for further explanations about the other algorithms we implemented we refer the reader to our previous work [33].

***Q*(λ)** In this algorithm, the acquired knowledge is stored in a table structure called Q-table [23]. This table stores pairs of states (states equal issues 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 [44] as follows:

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

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}, 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.

$Q(\lambda)$ 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 [23], 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 [33], 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 of (s, a).

The pseudocode depicted in Algorithm 1 is adapted from the one presented in chapter 12 in [23]. Regarding the above-mentioned 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.

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 //list of pairs of state-action visited
5:    s ← initial state s0 //being s a state
6:    while issues in model != Ø do //while issues left in the model
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 //reset table
11:          reset sae as an empty list //reset the list of pairs of state-action visited
12:       s_{t+1} ← a applied in s //next state comes from applying a in s
13:       Add (s,a) to sae list //add to state-actions visited
14:       e(s, a) ← e(s, a) + 1 //increment eligibility trace of the pair
15:       δ = r + γ max_{a'} Q(s_{t+1}, a') − Q(s, a) //Q(s,a) is the value of a pair in the Q-Table //a' is the best action for the next state
16:       for each s,a in sae do //for every pair visited
17:          Q(s, a) = Q(s, a) + αδe(s, a)
18:          e(s, a) ← γλe(s, a)
19:       t ← t + 1 //next step
20:       s ← s_{t+1} //s becomes next state
```

4.3.2 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 models are modified the better the performance might become since PARMOREL acquires and builds experience that is reused in later repairs.

To this extent, we define the *experience submodule*. This submodule makes use of the ML technique of transfer learning (TL) [19]. 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 [45]. For example, in the robot-in-the-maze problem, there might be a maze with water positions that should be avoided and other mazes where the robot should swim through the water.

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 [46]. 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(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.

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 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. The 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 starting-point method by copying all Q-table pairs of state-action without their Q-values, 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 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. More details about this process can be found in [19].

In traditional RL, the value of each entry in the Q-table (pairs 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 model repair, one entry'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 repair for one user might not be acceptable for another one. With TL, what is learnt from the repair of one model could be reused for other models. With this, consequent executions of PARMOREL, even by different users, could achieve better performance the more experience is reused.

We use the model in Fig. 6 to illustrate how PARMOREL supports TL. The learning information gained after each repair is represented by the concept of Experience which is composed of one to many entries and preferences. The concept Entry has references to all the elements that are part of the Q-table: an Issue and an Action. In addition, an Entry

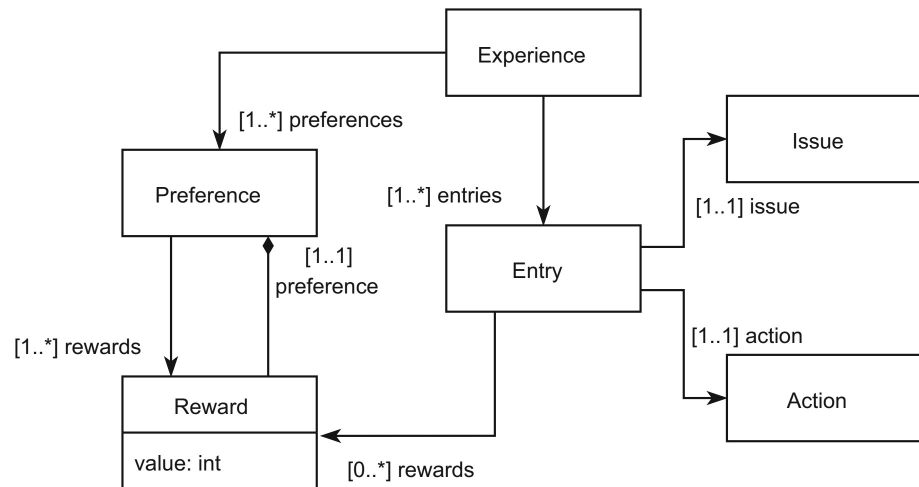


Fig. 6 Model of experience in PARMOREL

has 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 the following repairs. This way, if the current user shares any preference with previous ones, the rewards these previous preferences provided in previous repairs can be used to initialize the new user's Q-table, so that repair does not start from zero. This way, the learning will converge faster and fewer episodes will be required. When sharing experience in PARMOREL, we reduce the value of ϵ 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 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 constant during the execution otherwise some parts of the experience will be more favored than others.

An example of this process is displayed in Fig. 7. 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 repair a model with two issues, namely *issue1* and *issue2*. Both issues can be repaired with each of the actions *action1* and *action2*. Then, in the right part of Fig. 7 we show how the Q-table will look for *User2* once she starts using PARMOREL. 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 the discount factor. This way, when PARMOREL starts the repair 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.

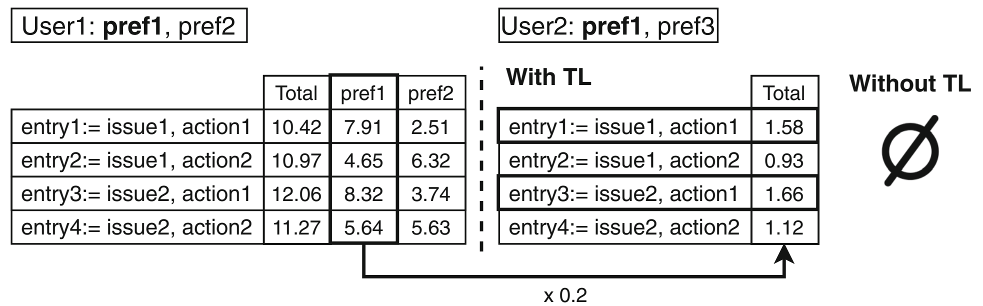
At the moment, the *experience submodule* is only supported when dealing with Ecore class diagrams. This is because the problems we have tackled so far in UML models (inconsistencies between class and sequence diagrams) are more dependent on each model instance structure (creating or moving messages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

For more details about how PARMOREL uses transfer learning and the *experience submodule* works, we refer the reader to our previous work [19,21].

5 Repairing syntactic errors

This section shows how PARMOREL can identify and repair syntactic errors in Ecore class diagrams. The *issues* and *actions submodule* are implemented using the syntactic errors implementation shown in Sect. 4.1.

The *preferences module* in this implementation combines two different techniques: a model distance metric to reward the preservation of the original model structure when repairing, minimizing undesired side-effects in the repaired model, and quality characteristics to reward quality improvement in the repaired model (see Sect. 4.2). In the following, we detail these two techniques before evaluating if PARMOREL can repair syntactic errors and produce results of better quality with this implementation.



	Total	pref1	pref2
entry1:= issue1, action1	10.42	7.91	2.51
entry2:= issue1, action2	10.97	4.65	6.32
entry3:= issue2, action1	12.06	8.32	3.74
entry4:= issue2, action2	11.27	5.64	5.63

	Total
entry1:= issue1, action1	1.58
entry2:= issue1, action2	0.93
entry3:= issue2, action1	1.66
entry4:= issue2, action2	1.12

Fig. 7 Example of TL between 2 users with a shared preference

5.1 Model distance

We have implemented the *preferences module* with a model distance calculator (see *Model distance* in Sect. 4.2) in order to understand how much a repaired model is close to the initial broken model. The model distance calculator is implemented as an Eclipse plugin, composed of a model matching algorithm specified with an ECL script [21]. The calculator obtains the distance between the initial broken model and the provisional repaired one produced in each episode and gives a distance reward to the *learning module*. Distance value goes from 0 to 1.0 with 1.0 meaning that the compared models are structurally the same, i.e., the closest distance possible.

This ECL script can be customized to add other constraints or relax the similarity function, e.g., remove the lower bound and upper bound matching for the structural features. This customization will affect the distance calculation and in turn, affects the repairing sequences chosen by PARMOREL. Indeed, by further restricting the comparison mechanism, the comparison algorithm will match elements differently.

To show how more refined distance calculation affects PARMOREL results, we have implemented two different distance metric calculations: *Basic* and *Custom*. The *Basic* matching algorithm was implemented for general purposes, and it uses the Levenshtein edit distance [47] when calculating the name similarity of different elements such as classes and structural features of the model. The *Custom* metric value is calculated using Eq. 2. The distance is computed as the sum of the class similarity ($classsim$) on the total number of classes ($nrclasses$) and features similarity ($featuresim$) on the number of features ($nrfeats$). For more details about the model distance calculator, we refer the reader to our previous work [48].

$$distance = ((((classsim)/nrclasses) + ((featuresim)/nrfeats))/2); \quad (2)$$

where *classsim* is a value calculated by an algorithm employing the Levenshtein edit distance [47] when calculating the name similarity of the classes, and similarly of the attributes and references *(featuresim).*

5.2 Quality characteristics

In this implementation, we specify the following quality characteristics to be used as user preferences: maintainability, understandability, complexity, and reusability (see *Quality characteristics* in Sect. 4.2). These characteristics are defined using a quality evaluation tool based on a definition of an EOL [49] script aggregating the available metrics in a predefined library [8].

The *maintainability* quality characteristic considered has been defined according to the definition given in [50] and the formula presented in [39], which is based on some of the metrics shown in Table 2 as follows:

$$Maintainability = \left(\frac{NC + NA + NR + DIT_{\mathrm{Max}} + \mathrm{Fanout}_{\mathrm{Max}}}{5}\right) \quad (3)$$

The definitions of the *understandability* and *complexity* quality characteristics are adopted from [51]. In particular, understandability can be defined as follows:

$$Understandability = \left(\frac{\sum_{k=1}^{NC} PRED + 1}{NC}\right) \quad (4)$$

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:

$$\mathrm{Complexity} = (NR - NUR + \mathrm{NOPR} + UND + (NR - \mathrm{NCR})) \quad (5)$$

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 Def. 4.

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 [52]. As presented in [53], *AIF* can be defined as follows:

$$Reusability = AIF = \left(\frac{INHF}{NTF}\right) \tag{6}$$

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

According to these definitions, 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 repairing sequences of actions, 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 add 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 Eq. 7 below). 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 Eq. 8). For example, by applying Eq. 7 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 Eq. 8 to these values, we obtain the x values 108 and 113.3. Finally, by applying Eq. 9, (where n is the number of quality characteristics selected by the user), we add all x values and we obtain the $reward$. At the moment, we consider all quality characteristics selected by the user will have the same priority, as prioritizing characteristics is not the focus of our study. A characteristic could be prioritized by multiplying each of its xs by a positive weight/integer instead of adding them directly in Eq. 9. Considering different priorities for the characteristics might be important depending on the domain where the repair is applied.

Table 2 Excerpt of the metrics used in the quality characteristics equations

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 predecessors in hierarchy	PRED

By doing this, the example repairing sequences of actions would get a reward of 117.3 for R1 and 221.3 for R2. Hence, PARMOREL would choose R2.

$$v = \begin{cases} (v_0 - v_r) + v_0, & \text{if decreasing characteristic} \\ v_r, & \text{if increasing characteristic} \end{cases} \tag{7}$$

$$x = \frac{v * 100}{v_0}, \quad \text{if } v == 0 \text{ then } x = 0 \tag{8}$$

$$reward = \sum_{i=1}^{n} x_i \tag{9}$$

5.3 Evaluation

Finally, we present an evaluation of the proposed implementation. In particular, we aim at answering the following research question:

RQ: How well can PARMOREL improve the precision in selecting better repaired models when the preference module is further extended?

5.3.1 Setup

Machine PARMOREL is run 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.

Table 3 Repairing models with error E11 while optimizing their complexity with the custom distance calculator

Model	Complexity		–
	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	=

Dataset For evaluating this implementation, we rely on the dataset from [24], mentioned in Sect. 2. We filtered the dataset in order to get only corrupted models, obtaining 107 models, where errors were distributed as in Table 1. We identified 12 error types, E1–E12, which were supported by PARMOREL. Table 1 details the total 973 error occurrences found in the dataset. The complete list and explanation of the errors and the corrupted models can be found at our GitHub repository[1].

5.3.2 Results and interpretation

To determine if a model is better than the rest we use quality characteristics. To better explain our results, first, we introduce how the considered quality characteristics are linked to model elements [50]. For instance, the maintainability of a model is influenced by the size of the model and then the number of classes; understandability is influenced by the 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 we have two classes with the same name in the model (E6 in Table 1), 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 with the customized one, 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 use as user preference, besides the distance calculation, to improve the complexity characteristic of the models. For this, we will repair models from the dataset containing error E11, a containment or bidirectional reference must be unique if its upper bound is different from 1 (see Table 1). 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).

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.

[1] https://github.com/models2020modelsrepair/ModelsRepair.git.

Table 4 Percentage of models which, after repairing with closest distance preference, are improved with respect to quality characteristics

Quality characteristic	Improved (%)
Maintainability	80.2
Reusability	84
Complexity	84
Understandability	100

Considering the custom distance algorithm extension as user preference, we proceed to measure if quality improved in the repaired models. The quality characteristics that are improved in relation to the whole dataset are reported[2] in Table 4. Maintainability has remained unchanged or improved in the 80.2% of the total models in the dataset[3], reusability in 84%, complexity in 84%, and understandability in 100%. 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, invalid specified literal (see Table 1), 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.

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 Eq. 3 are the 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 its equation (see Eq. 3).

Regarding reusability, it has improved in 84% of the cases. E8 is again the most present error in unimproved cases, and it does not affect the reusability equation (see Eq. 6), except when the faulty attribute 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 found on features which are not reflected in the quality calculation.

Finally, understandability is improved in all the models (100%). 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 answer our research question. By extending PARMOREL *preferences module*, the precision of the framework improves and it is able to produce repaired models with higher quality.

6 Restoring inter-model consistency

This section shows how PARMOREL can identify and restore inter-model consistency between UML class and sequence diagrams. The *issues* and *actions submodule* are implemented using the inconsistencies implementation shown in Sect. 4.1. The *preferences module* in this implementation uses a coupling calculation technique (see Sect. 4.2) in order to reward lower coupling in the sequence diagrams when restoring consistency.

In the following, we detail how PARMOREL detects and restores inconsistencies and calculates coupling before evaluating if it is able to restore inter-model consistency while lowering the coupling in the sequence diagrams.

6.1 Inconsistency rules and refactorings

To evaluate that PARMOREL is able to restore inter-model consistency, we have implemented the following rules (inspired by rules 110 and 114 from [37]) 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 a running example, we show in Fig. 8 a modification of the video on demand system (VoD) example presented in [54–56]. 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 triggers Rule 1. Additionally, the operation *loop* does not appear in any of the classes' lifelines in the sequence diagram, hence triggering

[2] The complete results are available as a Google spreadsheet in https://cutt.ly/zlnkPou.

[3] Some of the cases are excluded since the quality evaluation exited with errors or warnings.

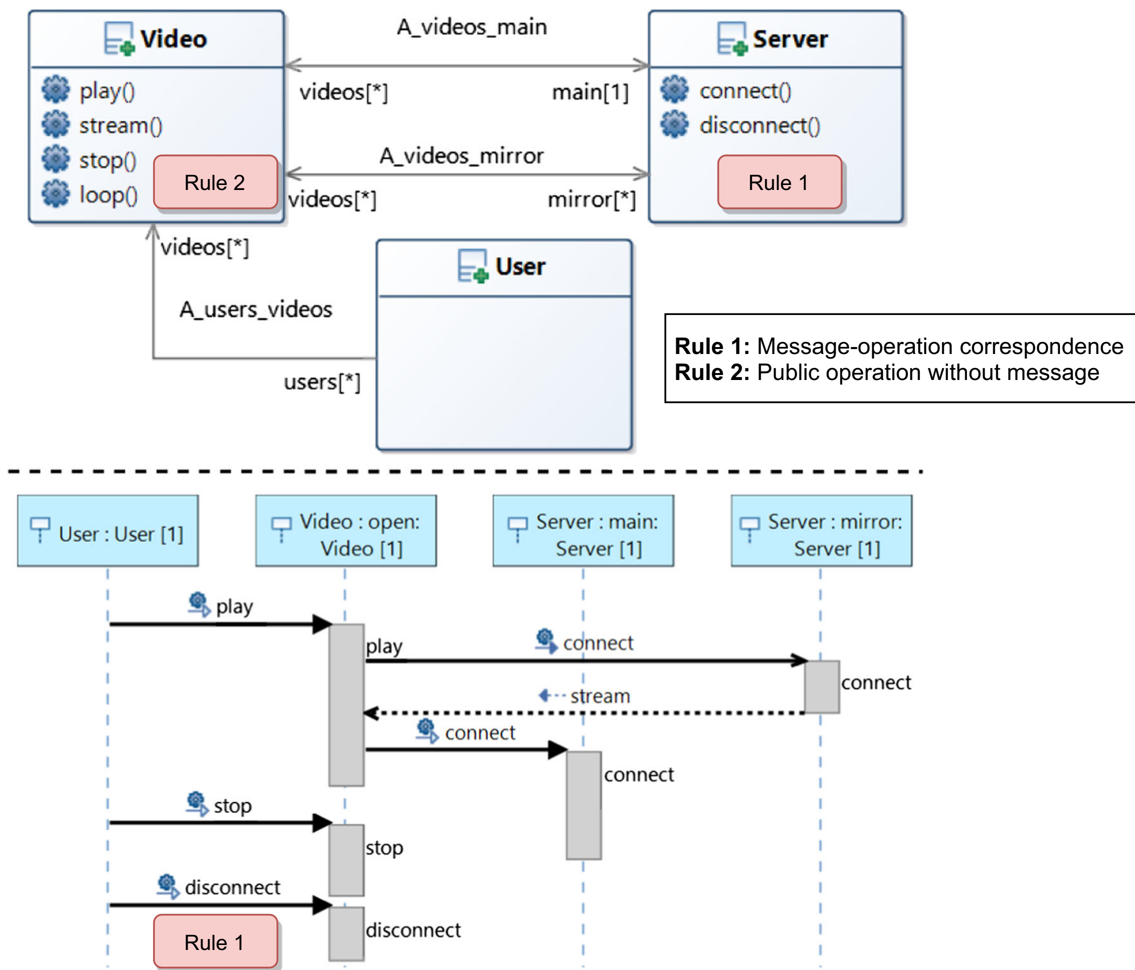


Fig. 8 Running example for inconsistencies between UML class and sequence diagrams

Rule 2. This operation should be invoked in *Video* either by *User* or *Server*.

To solve these inconsistencies, we have implemented two refactorings to be applied in the sequence diagrams:

- *Move message:* moves a message between lifelines to its corresponding place according to the class diagram, modifying the message's sender and/or receiver. It solves inconsistencies caused by violating Rule 1.
- *Add message:* adds a message in a lifeline (receiver) according to its corresponding operation and the class it belongs to in the class diagram. It solves inconsistencies caused by violating Rule 2.

Although each refactoring addresses inconsistencies caused by violating one of the rules, PARMOREL has to decide where to add or move the messages, since depending on where the messages are located, the overall coupling of the sequence diagram will be different. The objective here is to keep the coupling as low (or loose) as possible.

6.2 Coupling calculation

As explained in Sect. 4.2, by using the metrics offered in SDMetrics [36], we can define preferences to guide the repair of UML models. SDMetrics offers a suite of predefined and extensible metrics.

As an example, we use the predefined metrics MsgSent (number of messages sent) and MsgRecv (number of messages received) inspired by [43] to calculate the coupling in UML sequence diagrams. By taking into account the number of messages each class lifeline in the sequence sends and receives, we can measure the degree of interdependence between the classes in the sequence diagram. By adding the number of messages sent and received we can obtain the coupling of each class. This way, PARMOREL can decide which lifelines will be the best to act as senders and receivers to keep the coupling as low as possible.

To do so, we sum up the values of MsgSent and MsgRecv of each lifeline (being n the total number of lifelines, see Eq. 10). 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 Eq. 11). This value will be higher the lower the total coupling of the diagram. We use this value as a reward for the learning algorithm. Since PARMOREL works with an ϵ-greedy algorithm, it will always look to maximize the rewards.

$$sum = \sum_{i=1}^{n} MsgSent_i + MsgRecv_i \tag{10}$$

$$reward = \sum_{i=1}^{n} \frac{sum}{MsgSent_i + MsgRecv_i} \tag{11}$$

For example, returning to the example in Fig. 8, 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 has a reward of 26.5; the solution is displayed in Fig. 9. It 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 receiver and *Video* as sender would get a reward of 20.66. In this case, both *Video* and *Server:main* lifelines would have worse coupling.

This coupling preference is implemented as an example to test the extensibility of PARMOREL preferences and personalization of repair when solving inter-model inconsistencies in UML models. We consider the repair is correct when none of the rules we defined are violated. However, this same scenario could be solved with other preferences, such as cohesion or other quality characteristics.

6.3 Evaluation

Finally, we present an evaluation of the proposed implementation to restore inter-model consistency. First, we present our experiment's setup, followed by the results obtained and their interpretation. In particular, we aim at answering the following research question:

RQ: How well can PARMOREL provide personalized restoration of inter-model consistency between UML class and sequence diagrams?

6.3.1 Setup

Machine PARMOREL is run in UML Designer 9.0 (Obeo) on the same laptop presented in Sect. 5.3.

Dataset To test this implementation, we have manually created 12 models, 6 pairs of class and sequence diagrams using the Eclipse IDE of UMLDesigner 9.0 [4]. The sequence diagrams are based on sequence diagrams that can be found in [57]. Regarding the class diagrams, we created them based on these sequence diagrams, since in [57] they were not available. Furthermore, we arbitrarily changed the sequence diagrams so that inter-model inconsistencies would appear between them and the class diagrams, as the diagrams available did not contain this kind of issues. This dataset is available to download at [34].

The subject sequence diagrams have between 4 and 11 lifelines and include between 2 and 10 violations of rules 1 and 2.

6.3.2 Results and interpretation

In this experiment, we evaluate if 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 lifelines could be potential senders and receivers for the repair actions. 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.

As an answer to our research question, PARMOREL is able to resolve all inconsistencies in the dataset models, always choosing the most optimal solution with respect to coupling (for the given formulae which are used to calculate the coupling, i.e., the repaired sequence diagrams might not be the most optimal representation of their domain). This solution is the sequence of actions that create the model with 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 experiment, we have discovered that PARMOREL is able to design a sequence diagram from scratch. When this happens, for every operation existing in the class diagram, a violation of Rule 2 is triggered, since there are no messages existing in the sequence

[4] http://www.umldesigner.org/.

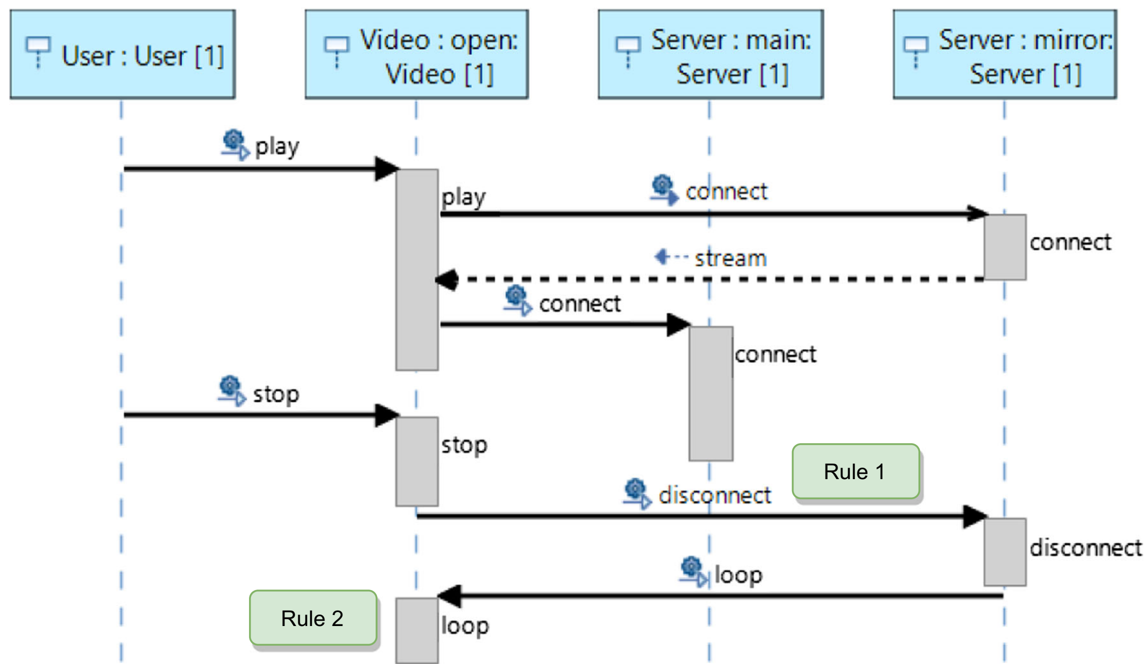


Fig. 9 PARMOREL's solution for running example in Fig. 8

diagram. Hence, PARMOREL is able to find which is the most optimal distribution of messages between the lifelines in the sequence diagram to reduce the coupling and create a diagram with such distribution.

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.

7 Threats to validity

In this section, we discuss potential threats that are associated with the validity of the experiments discussed in Sects. 5 and 6. We distinguish between internal and external threats to validity.

7.1 Internal validity

Internal threats are factors influencing the outcomes of the performed experiments, hence, we present the internal validity threats for each of our experiments.

PARMOREL 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, ignoring the issue and continuing with the repair process. Hence, the repair is dependent on the actions provided in the actions submodule. We plan to address this as part of our future work.

No real users participated in the experiments. However, the preferences that guide the repair in both experiments simulate the decisions that could have been taken by a real user.

Among the internal threats, we have the quality evaluation process since the quality model is user-defined. Quality characteristics are often based on the modeler's experience and mistakes in these quality models' definitions may impact the results. To mitigate these issues we reused, when possible, existing definitions from literature and represented them faithfully with the corresponding models or DSL syntax.

Quality of the repaired models is measured through metrics and not evaluated by users. Although a metric cannot include the fine-grained details that a real user could evaluate in the model, we consider these metrics as a good indicator of the objective quality of the repaired model.

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.

In this direction, the coupling metric is also user defined. To mitigate this threat, we followed definitions implemented in SDMetrics, based on formulas extracted from the literature.

As stated in Sect. 6.2, the coupling preference is used as an example to prove PARMOREL's extensibility. However, when solving inter-model inconsistencies, taking into account the models' representation quality—i.e., to which extend the models represent the domain sufficiently—could lead to results more aligned with the user intentions.

7.2 External validity

In this context, we discuss how the conducted experiments would still be valid outside the used setting. To mitigate this aspect, in Experiment 1 (see Sect. 5) we considered various models since the dataset is heterogeneous and used in other experiments in literature [58].

For Experiment 2 (see Sect. 6), the number of models is not large for a standard evaluation but finding real corrupted models with inter-model inconsistencies on existing repositories is not easy. Usually repositories consist of isolated models. However, this threat is justified by the scope of the experiment and the fact that the subject models were inspired by real ones (genMyModel repository). Another solution would have been to create a synthetic dataset extracted, for example, from real project via reverse engineering. We plan to create such a dataset as part of our future work.

Also in Experiment 2, we chose inter-model inconsistency between class and sequence diagrams as an example to show PARMOREL's potential to address this type of issue. However, inter-model inconsistencies can happen in many different types of models. We consider this example representative enough as a first step of PARMOREL dealing with this type of issue.

Throughout the paper, we have picked four quality characteristics as a proof of concept to measure the quality of the refactored model (maintainability, understandability, reusability, and complexity), and with a coupling and model distance measure.

Many other characteristics could be measured in the models and, other issues could be identified together with different refactorings and 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. Also, including experts in the quality definition process could mitigate this aspect.

Finally, the examples in the paper are based on EMF, Ecore, and UML models (class and sequence diagrams), but as we explained, it is possible to switch to other model types by extending PARMOREL. Within EMF, the work presented in this paper is specific for Ecore class diagrams. 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 advantage that RL presents with respect to other approaches that solve similar problems, such as search-based approaches or multi-objective heuristics is the ability to learn. RL's objective is not only to solve a problem but to learn how to solve it. When facing new problems, the algorithm will not start from scratch and repair time can be streamlined.

The most similar work to ours we could find is [59], 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. [6] 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 [60] 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. [4] 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 [61] 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 algo-

rithms 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.

Some approaches make use of neural network (NN) architectures to solve different MDE problems. In [31], authors present a NN architecture for model transformation without specifying code for any specific transformations. Tackling model refactoring, in [62] authors make use of a deep NN architecture to refactor UML diagrams with symptoms of design flaws. NNs need a great amount of data in order to work. The solutions are tightly related to the training dataset, so if the requirements of the problem change, 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.

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 [63] 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. 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 [64]. In this work, the 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 [65]. 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 modules: modeling, preferences and learning modules. 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 support the concepts of states (issues in our problem), actions and rewards [23].

Throughout the experiments presented in Sects. 5, and 6 we have demonstrated how we have extended PARMOREL's modules with the implementations presented in Sect. 4. We have evaluated PARMOREL's ability to repair different types of models (e.g., Ecore class diagrams, UML class diagrams, and UML sequence diagrams), different types of issues (e.g., syntactic errors, and inter-model inconsistencies), and with different user preferences (e.g., model distance, and coupling).

Our experiments show that PARMOREL can repair model issues by choosing the optimal action from a set of available ones (Sect. 5), Additionally, we have discovered that PARMOREL could be used to assist modelers—with some degree of auto-completion—when designing inter-related models (Sect. 6).

Our evaluation shows satisfactory and promising results. We find that PARMOREL is an indicator of how ML could be useful within the model repair field and, that it could be used as a suit where different model repair experiments could be performed in a single environment, without the need of using various tools and the nuisances caused by integrating them.

In the future, we plan to extend PARMOREL so that models without issues could be improved in terms of, for example, quality characteristics. Next, we plan to create a benchmark using different model datasets, including the ones used in this paper, with which we will compare PARMOREL results and its performance to other existing model refactoring and repair approaches in the literature. We would like to improve our inter-model inconsistencies with synthetic models. Also, we plan to extend PARMOREL to solve other problems relevant in the modeling field, like model instances refactoring after their corresponding metamodel evolves, making architectural models compliant to best practice patterns, repair of behavioral models and taking into account execution semantics.

Additionally, we plan to extend the learning module with other algorithms beyond reinforcement learning, especially focusing on other AI approaches, and studying their performance with respect to RL algorithms. Also, we will compare the tool performance with other automatic repairing tools presented in Sect. 8, paying special attention to search-based, rule-based, and automated planning approaches.

Moreover, we plan to extend the experience submodule so that transfer learning can be applied to more complex repair problems such as restoring inter-model consistency. Lastly, we plan to extend PARMOREL's actions submodule so that repair actions can be directly inferred from the issues detected in the models.

Funding Open access funding provided by Western Norway University Of Applied Sciences.

References

1. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Quality-driven detection and resolution of metamodel smells. IEEE Access **7**, 16364–16376 (2019)
2. Strittmatter, M., Hinkel, G., Langhammer, M., Jung, R., Heinrich, R.: Challenges in the evolution of metamodels: smells and anti-patterns of a historically-grown metamodel (2016)
3. Feldmann, S., Kernschmidt, K., Wimmer, M., Vogel-Heuser, B.: Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. J. Syst. Softw. **153**, 105–134 (2019)
4. Taentzer, G., Ohrndorf, M., Lamo, Y., Rutle, A.: Change-preserving model repair. In: International conference on fundamental approaches to software engineering, pp. 283–299. Springer (2017)
5. Ohrndorf, M., Pietsch, C., Kelter, U., Kehrer, T.: Revision: a tool for history-based model repair recommendations. In: Proceedings of the 40th International conference on software engineering: companion proceeedings, pp. 105–108. ACM (2018)
6. Nassar, N., Radke, H., Arendt, T.: Rule-based repair of EMF models: An automated interactive approach. In: International conference on theory and practice of model transformations, pp. 171–181. Springer (2017)
7. Macedo, N., Guimaraes, T., Cunha, A.: Model repair and transformation with echo. In: Proceedings of the 28th IEEE/ACM International conference on automated software engineering, pp. 694–697. IEEE Press (2013)
8. Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: A tool-supported approach for assessing the quality of modeling artifacts. J. Comput. Lang. **51**, 173–192 (2019)
9. López-Fernández, J.J., Guerra, E., De Lara, J.: Assessing the quality of meta-models. In: MoDeVVa@ MoDELS, pp. 3–12. Citeseer (2014)
10. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. In: Proceedings of the 2nd international conference on Software engineering, pp. 592–605. IEEE Computer Society Press (1976)
11. Dromey, R.G.: A model for software product quality. IEEE Trans. Software Eng. **21**(2), 146–162 (1995)
12. Ortega, M., Pérez, M., Rojas, T.: Construction of a systemic quality model for evaluating a software product. Software Qual. J. **11**(3), 219–242 (2003)
13. Williams, J.R., Zolotas, A., Matragkas, N.D., Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: What do metamodels really look like? Eessmod@ Models **1078**, 55–60 (2013)
14. Khelladi, D.E., Kretschmer, R., Egyed, A.: Detecting and exploring side effects when repairing model inconsistencies. In: Proceedings of the 12th ACM SIGPLAN international conference on software language engineering, pp. 113–126 (2019)
15. Addazi, L., Cicchetti, A., Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Semantic-based model matching with emfcompare. In: ME@ MODELS, pp. 40–49 (2016)
16. Kehrer, T., Kelter, U., Taentzer, G.: 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), Birkhäuser, Cham, pp. 163–172. IEEE (2011)
17. Syriani, E., Bill, R., Wimmer, M.: Domain-specific model distance measures. J. Object Technol. **18**(3), 3 (2019)
18. Barriga, A., Rutle, A., Heldal, R.: 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, pp. 175–181 (2019). https://doi.org/10.1109/MODELS-C.2019.00030
19. Barriga, A., Rutle, A., Rogardt, H.: Improving model repair through experience sharing. J. Object Technol. **19**(2), 13:1-21 (2020). https://doi.org/10.5381/jot.2020.19.2.a13
20. Iovino, L., Barriga, A., Rutle, A., Rogardt, H.: Model repair with quality-based reinforcement learning. J. Object Technol, Elsevier, 2014 **19**(2), 17:1-21 (2020). https://doi.org/10.5381/jot.2020.19.2.a17
21. Barriga, A., Heldal, R., Iovino, L., Marthinsen, M., Rutle, A.: 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 (2020)
22. Barriga, A., Bettini, L., Iovino, L., Rutle, A., Heldal, R.: Addressing the trade off between smells and quality when refactoring class diagrams. J Object Technol. **20**(3), 1:1-15 (2021). https://doi.org/10.5381/jot.2021.20.3.a1
23. Thrun, S., Littman, M.L.: Reinforcement learning: an introduction. AI Mag. **21**(1), 103–103 (2000)
24. Önder Babur: A labeled Ecore metamodel dataset for domain clustering (2019). https://doi.org/10.5281/zenodo.2585456
25. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Automated classification of metamodel repositories: a machine learning approach. In: 2019 ACM/IEEE 22nd International conference on model driven engineering languages and systems (MODELS), pp. 272–282. IEEE (2019)
26. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education, London (2008)
27. Beck, K., Fowler, M., Beck, G.: Bad smells in code. Refact. Improv. Des. Exist. Code **1**, 75–88 (1999)
28. Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Softw. Syst. Model. **19**(1), 5–13 (2020)
29. Shafiq, S., Mashkoor, A., Mayr-Dorn, C., Egyed, A.: Machine learning for software engineering: a systematic mapping. arXiv preprint arXiv:2005.13299 (2020)
30. Cabot, J., Clarisó, R., Brambilla, M., Gérard, S.: Cognifying model-driven software engineering. In: Federation of international conferences on software technologies: applications and foundations, pp. 154–160. Springer (2017)
31. Burgueño, L., Cabot, J., Gérard, S.: 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 (2019)
32. Ghannem, A., El Boussaidi, G., Kessentini, M.: Model refactoring using interactive genetic algorithm. In: International symposium on search based software engineering, pp. 96–110. Springer (2013)
33. Barriga, A., Mandow, L., Perez de la Cruz, J.L., Rutle, A., Heldal, R., Iovino, L.: 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) (2020). To appear
34. Project PARMOREL, Last accessed on 19/05/2021, https://ict.hvl.no/project-parmorel/
35. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta: An approach for defining and applying reusable metamodel refactorings. In: MODELS (Satellite Events), pp. 71–80 (2017)
36. Wust, J.: Sdmetrics: The software design metrics tool for uml (2005)
37. Torre, D., Labiche, Y., Genero, M., Elaasar, M.: A systematic identification of consistency rules for uml diagrams. J. Syst. Softw. **144**, 121–142 (2018)
38. Iovino, L., Barriga, A., Rutle, A., Rogardt, H.: Model repair with quality-based reinforcement learning. J. Object Technol. **19**(2), 17 (2020)

39. Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: 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 (2016)
40. Lopes, D., Hammoudi, S., De Souza, J., Bontempo, A.: Metamodel matching: Experiments and comparison. In: 2006 International conference on software engineering advances (ICSEA'06), pp. 2–2. IEEE (2006)
41. Gray, J., Rumpe, B.: Conceptual distance of models and languages (2019)
42. Kolovos, D., Rose, L., Paige, R., Garcıa-Domınguez, A.: The epsilon book. Structure **178**, 1–10 (2010)
43. Briand, L., Devanbu, P., Melo, W.: An investigation into coupling measures for c++. In: Proceedings of the 19th international conference on Software engineering, pp. 412–421 (1997)
44. Bellman, R.: Dynamic Programming. Courier Corporation (2013)
45. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Trans. Knowl. Data Eng. **22**(10), 1345–1359 (2010)
46. Torrey, L., Shavlik, J.: Transfer learning. In: Handbook of research on machine learning applications and trends: algorithms, methods, and techniques, pp. 242–264. IGI Global (2010)
47. Levenshtein, V.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady **10**, 707 (1966)
48. Di Ruscio, D., Iovino, L., Pierantonio, A., Bettini, L.: Detecting metamodel evolutions in repositories of mde projects. In: Modelling Foundations and Applications. Springer (2020). To appear
49. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon object language (eol). In: European conference on model driven architecture-foundations and applications, pp. 128–142. Springer (2006)
50. Genero, M., Piattini, M.: 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)
51. Sheldon, F.T., Chung, H.: Measuring the complexity of class diagrams in reverse engineering. J. Softw. Maint. **18**(5), 333–350 (2006). https://doi.org/10.1002/smr.336
52. Arendt, T.: Quality assurance of software models - A structured quality assurance process supported by a flexible tool environment in the eclipse modeling project. Ph.D. thesis, University of Marburg (2014). http://archiv.ub.uni-marburg.de/diss/z2014/0357
53. Al-Ja'Afer, J., Sabri, K.: Metrics for object oriented design (mood) to assess java programs. Tech. rep., King Abdullah II school for information technology, University of Jordan, Jordan (2007)
54. Ohrndorf, M., Pietsch, C., Kelter, U., Grunske, L., Kehrer, T.: History-based model repair recommendations. ACM Trans. Softw. Eng. Method. **30**(2), 1–46 (2021)
55. Egyed, A.: Instant consistency checking for the uml. In: Proceedings of the 28th international conference on Software engineering, pp. 381–390 (2006)
56. Macedo, N., Jorge, T., Cunha, A.: A feature-based classification of model repair approaches. IEEE Trans. Software Eng. **43**(7), 615–640 (2016). https://doi.org/10.1109/TSE.2016.2620145
57. Dirix, M., Muller, A., Aranega, V.: Genmymodel: an online uml case tool (2013)
58. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Automated classification of metamodel repositories: a machine learning approach. In: 2019 ACM/IEEE 22nd International conference on model driven engineering languages and systems (MODELS), pp. 272–282 (2019)
59. Puissant, J.P., Van Der Straeten, R., Mens, T.: Resolving model inconsistencies using automated regression planning. Softw. Syst. Model. **14**(1), 461–481 (2015)
60. Cervantes, A.A., van Beest, N.R., La Rosa, M., Dumas, M., García-Bañuelos, L.: Interactive and incremental business process model repair. In: OTM Confederated international conferences "on the move to meaningful internet systems", pp. 53–74. Springer (2017)
61. Kretschmer, R., Khelladi, D.E., Egyed, A.: An automated and instant discovery of concrete repairs for model inconsistencies. In: Proceedings of the 40th international conference on software engineering: companion proceeedings, pp. 298–299. ACM (2018)
62. Sidhu, B.K., Singh, K., Sharma, N.: A machine learning approach to software model refactoring. Int. J. Comput. Appl. **44**, 166 (2020)
63. Kessentini, M., Mansoor, U., Wimmer, M., Ouni, A., Deb, K.: Search-based detection of model level changes. Empir. Softw. Eng. **22**(2), 670–715 (2017)
64. Moghadam, I.H., Ó Cinnéide, M.: Code-imp: a tool for automated search-based refactoring. In: Proceedings of the 4th workshop on refactoring tools, pp. 41–44 (2011)
65. Selman, B., Gomes, C.P.: Hill-climbing search. Encyclopedia of cognitive science (2006)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Angela Barriga holds a PhD in Computer Science from the 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, especially 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.

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.

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. 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 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.

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 committees 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.