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

Improving model repair through experience sharing

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

a. Western Norway University of Applied Sciences, Norway

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

Keywords Model Repair; Reinforcement Learning; Transfer Learning

1 Introduction

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

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

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

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

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

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

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

2 Motivation

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

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

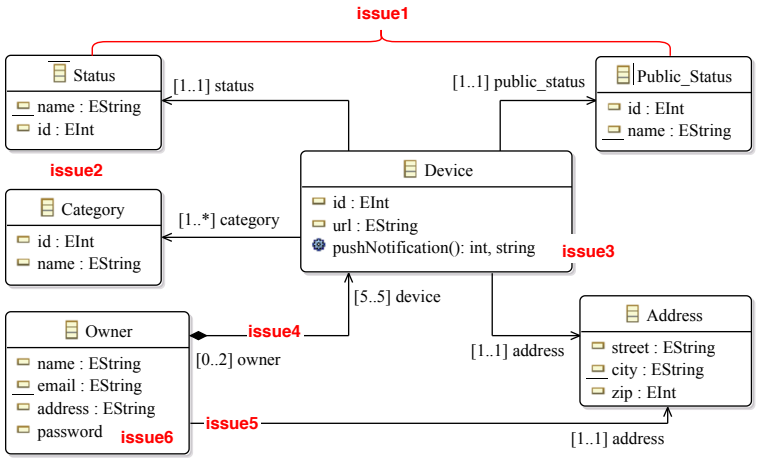

Figure 1 – Sample model containing a variety of issues

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

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

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

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

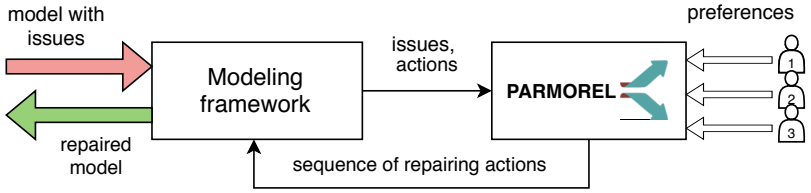

Figure 2 – Overview of our approach

3 Reinforcement Learning in PARMOREL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algorithm 1 Q-learning in PARMOREL

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

4 Applying transfer learning in model repair

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

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

4.1 Learning through propagating preferences

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

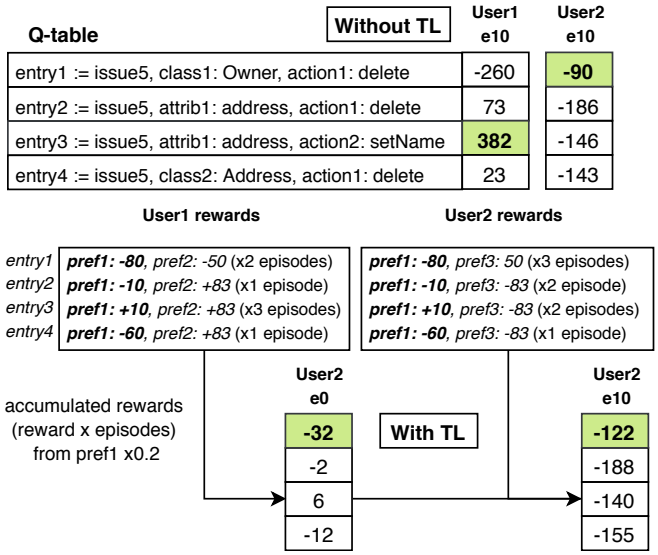

Figure 4 – Example of differences when initializing the Q-table with and without TL

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

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

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

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

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

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

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

4.2 Integration with PARMOREL

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

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

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

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

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

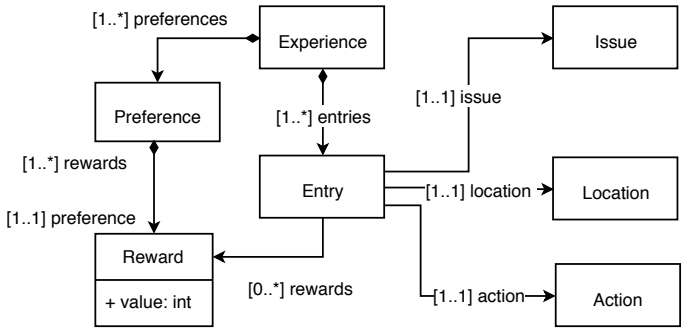

Figure 5 – Model of transferring learning experience in PARMOREL

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

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

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

Algorithm 2 Transfer learning in PARMOREL

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

5 Implementation and Evaluation

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

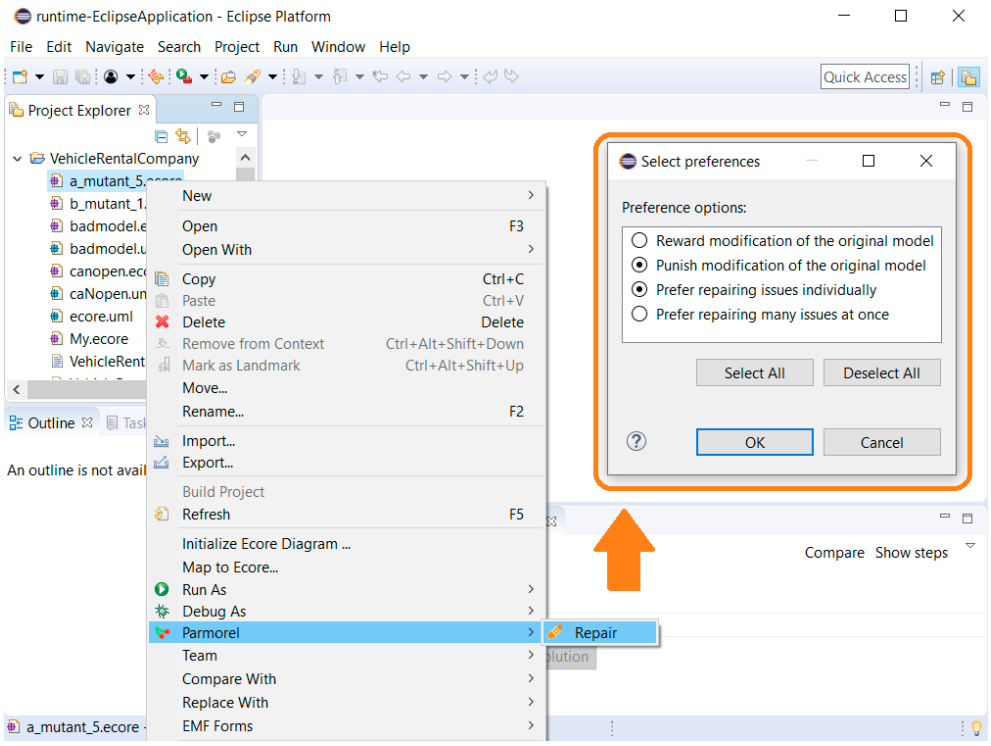

Figure 6 – Screenshot of PARMOREL Eclipse plugin

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

5.1 Example I: different users repairing the same model

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

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

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

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

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

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

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

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

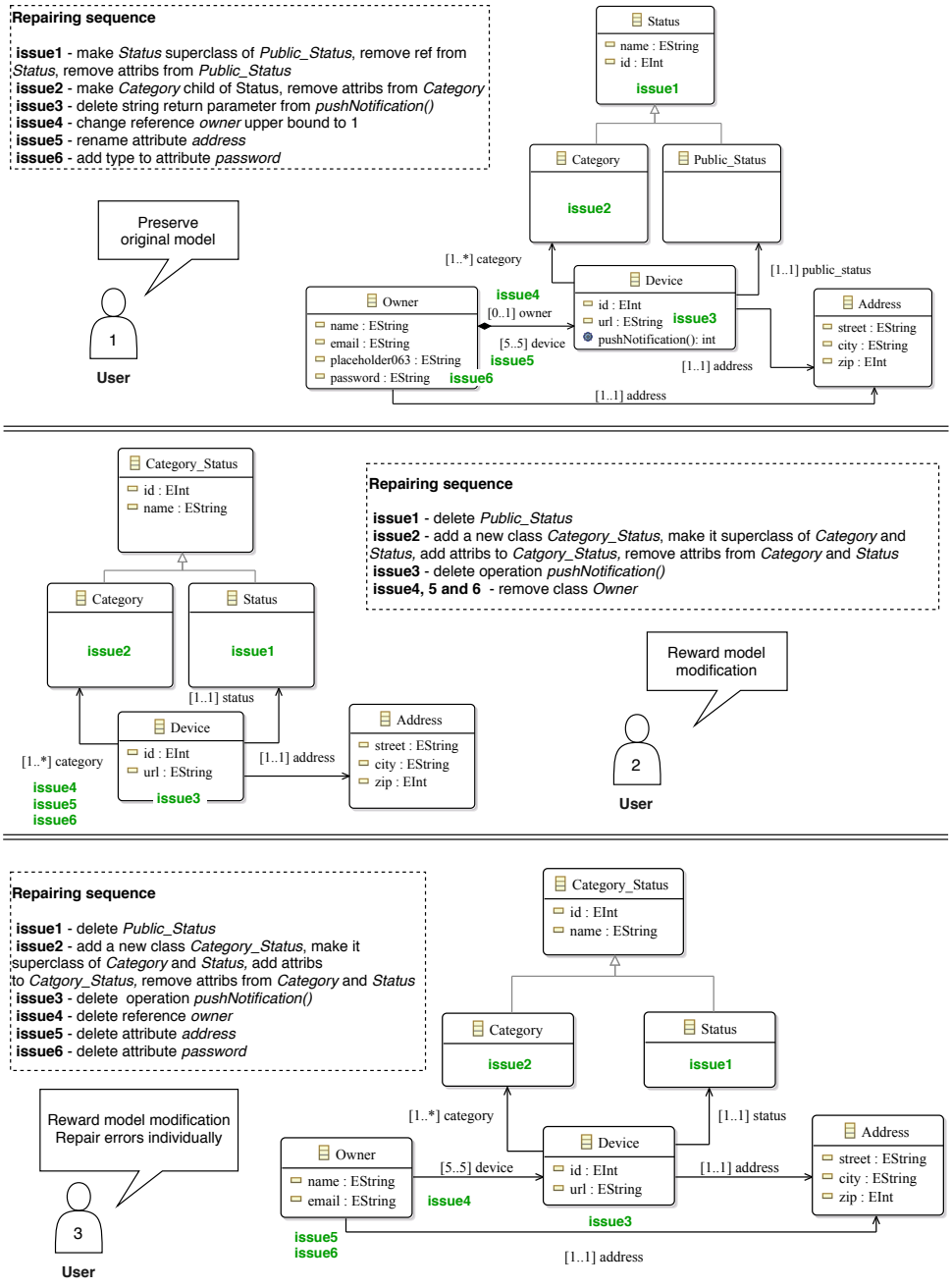

Figure 7 – Users with di□erent preferences repair the same broken model

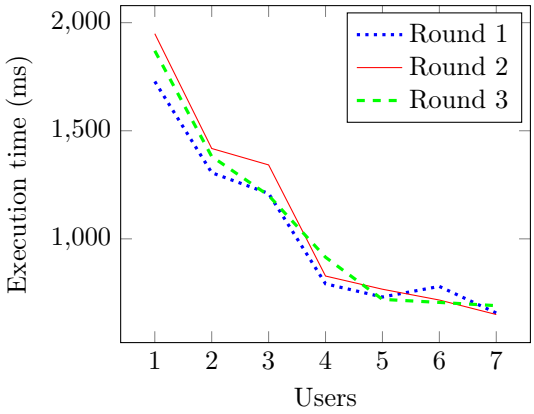

Figure 8 – Evolution of repair times for model in Fig. 1 with 7 users in 3 di□erent rounds

5.2 Example II: di□erent users repairing randomly mutated models

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

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

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

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

Table 1 – Comparison of repairing times with and without TL

6 Threats to validity

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

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

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

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

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

7 Related work

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

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

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

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

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

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

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

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

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

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

8 Conclusions and future work

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

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

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

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

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

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

References

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

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

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

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

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

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

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

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

[BRH19] Angela Barriga, Adrian Rutle, and Rogardt Heldal. Personalized and automatic model repairing using reinforcement learning. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 175–182, 2019. [Forthcoming]. Available: https://bit.ly/2IPfwMD.

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

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

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

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

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

[KKE19] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. Detecting and exploring side effects when repairing model inconsistencies. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–126, 2019.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

About the authors

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

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

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