Personalized and automatic model repairing using reinforcement learning

Angela Barriga
Department of Software Engineering
Western Norway University
of Applied Sciences
Bergen, Norway
abar@hvl.no

Adrian Rutle
Department of Software Engineering
Western Norway University
of Applied Sciences
Bergen, Norway
aru@hvl.no

Rogardt Heldal
Department of Software Engineering
Western Norway University
of Applied Sciences
Bergen, Norway
rohe@hvl.no

***Abstract*—When performing modeling activities, the chances of breaking a model increase together with the size of development teams and number of changes in software specifications. Model repair research mostly proposes two different solutions to this issue: fully automatic, non-interactive model repairing tools or support systems where the repairing choice is left to the developer's criteria. In this paper, we propose the use of reinforcement learning algorithms to achieve the repair of broken models allowing both automation and personalization. We validate our proposal by repairing a large set of broken models randomly generated with a mutation tool.**

***Index Terms*—Model Repair, Reinforcement Learning, Personalization**

I. Introduction

Models are often used to develop key parts of systems in engineering domains [1]. Correctness and accuracy of such models are important to produce the systems they represent. The difficulty of keeping models free of errors grows proportionally with models size, complexity and number of changes introduced during their lifetime. Tools that automatize or support error detection and repairing of models can improve how organizations deal with model-driven engineering processes by reducing the burden of manually dealing with correctness issues, improving delivery time and final quality.

Despite the benefits aforementioned, automatic repairing has the drawback of providing the same solution for a certain error while each modeler may have different preferences for repairing it. Hence, proper model repair would not be feasible without finding a balance between automation and personalization of repair [2]. Our approach permits personalization for each modeler so that an error can be repaired in different ways.

We propose reinforcement learning (RL) as a solution for this challenge [3]. 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 [4]. RL provides the necessary structure to adapt to different personalization settings and to perform better after each execution. The contributions of this paper are (i) an approach to apply RL for model repair, and (ii) a proof of concept implementation.

II. Background

This section introduces some basic theoretical notions of RL and Q-Learning in order to provide a comprehensive guide to understand the rest of this paper.

The authors in [5] stress how the combination of machine learning (ML) and modeling could be beneficial for the future of the modeling domain. In particular, ML could reduce the amount of time spent in repairing broken models and improve its quality once repaired. Many well-known ML algorithms depend on large amounts of data to learn how to repair a problem [6]. This is a challenge in the modeling domain since available model repositories (like [7], [8]) only offer data limited in terms of size and diversity.

We propose RL because a training phase and labelled data are no longer needed; learning is achieved from the interaction between a learning agent and its environment (see Fig. 1). The agent performs actions in the environment that changes its state. Each action gets a reward depending on the state produced. Performance will be poor at the beginning, improving as the agent gains experience. 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 to a free space), it receives a reward, contrarily (stepping on a wall) it is penalized. The agent continues performing actions, seeking the highest reward until it reaches its ultimate goal; e.g., the exit of the maze.

RL is a field with many different algorithms, we choose Q-learning [4] as a proof of concept to apply RL to repair models. The reasoning behind this decision is the simplicity to deal with diverse information due to its table nature. In Q-Learning, knowledge is stored in a structure called Q-table, which is easy to export and import into new executions and keeps data in a generic format. The agent chooses then the most optimal action available by consulting the Q-table. This table is initialized with zeros when the algorithm begins, and it is updated while the agent interacts with the environment with repeated calculations based on the Bellman Equation [9]:

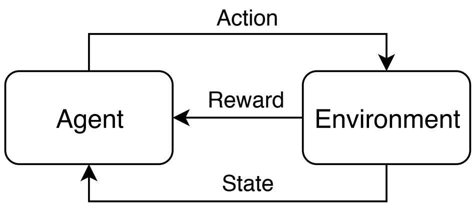

Fig. 1: Agent-environment interaction in RL

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

This equation returns a weight, called *Q-value*, telling that the maximum future reward is the reward r the agent received for entering the current state s_t with some action a_t plus the maximum future reward for the next state s_{t+1} and action a_{t+1} reduced by a discount factor γ to avoid falling in a local maximum. This allows to infer the value of the current state s_t based on the calculation of the next one s_{t+1}, which can be used to calculate an optimal policy to select actions (while future states are not known in a real system, we can obtain them by repairing current state s_t with a_t and checking which is the next error s_{t+1} to repair, since we can obtain all errors present in a faulty model at any repairing time t). The factor α provides the learning rate, which determines how much Q-values change from one iteration to the next one. From here, the algorithm can determine the optimal action to apply to the next state until it reaches its final goal; this process is known as exploitation, and this calculation is repeated every time t the algorithm selects an action.

One of the variables used to calculate the Q-value, is the maximum weight stored in the Q-table for the next error to repair ($\max_a Q_t(s_{t+1}, a_{t+1})$). This allows us to measure the consequences of applying a certain action in the model (e.g., if applying an action creates a new unknown error this action will be punished, getting a lower weight).

Additionally, the algorithm can pick random actions instead of the one with biggest reward in order to find new and perhaps more optimal solutions; this process is called exploration [10]. By combining both exploitative and explorative policies, RL can provide a wider range of solutions.

Each iteration in which the agent performs actions to achieve the final goal is called an episode. Inside the episode, the agent performs several steps attempting to find the best action for the current state. For every action performed in a state, the agent updates the values of the Q-table for that pair of state and action. The number of episodes and steps are decided depending on the size of the problem to solve (i.e., the number of states and actions) so that the agent can get enough time to reach its final goal. The Q-table can be exported and later imported in new executions, this way the agent will gain experience and will perform better with time.

III. Model Repair

In this section, we present our approach for using RL in model repair. At the moment, we repair well-formedness errors

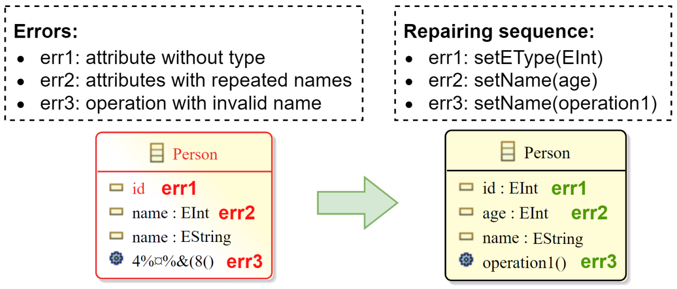

Fig. 2: Repairing of 3 errors in a sample class

in class diagram models. To perform the repair, first, we need a modeling framework able to provide us with errors present in a model, every context where each error can be repaired and a list of actions to modify the model. In our implementation, we obtain them from the Eclipse Modeling Framework (EMF) [11]. Therefore, the errors the approach is able to repair correspond to the catalogue of errors provided by the EMF diagnostician.

Regarding actions, first, the algorithm filters actions invokable in each error's contexts (e.g., for an error in an attribute, only actions regarding attributes). Then, actions are filtered again, maintaining only those which are able to repair any of the errors. The EMF diagnostician provide contexts organized in a hierarchy, ordered by their level in the model (e.g., a class has a higher level than attributes and operations, which belong to classes). For example, err2 in Fig. 2 (attributes with repeated names) can be repaired at the class (Person) by deleting it, or by deleting or renaming any of the faulty attributes. This makes 3 possible repairing contexts, being class (Person) the highest, attribute1 (name) the middle, and attribute2 (name) the lowest. This error, (err2) will always keep these exact contexts (class - attribute1 - attribute2), independent of the model in which it appears. When contexts belong to the same level of the model, like attributes in this example, the hierarchy order is based on their spatial arrangement (attribute1 appears above attribute2).

Based on this example, a starting point for err2 in our RL algorithm would be the Q-table showed in Table I, with all weights initialized to zero (values showed in Table I are calculated during the repairing). The Q-table adapted to model repair is a 3-dimensional structure, storing a weight for a combination of error, context and action. Table I contains only those actions able to repair err2 in any of its contexts, other actions are discarded.

Our use of RL is slightly different from the standard. In the example of the robot and the maze (see Section II), the final goal is finding the exit of the maze and the robot performs actions towards it. In our case, we aim to completely repair the model, but our RL algorithm must repair one error before moving to the next one. By targeting one error at a time, we reduce the state space of the algorithm. Hence, while the robot does not know about the states of the maze, we know the errors that are present in the model (our states). In our model repair process, the objective is not just to repair the model, but to

Error	Context	Action	Weight
Attributes with repeated names	class (Person)	delete	-100
	attribute1 (name)	setName	300.19
		delete	-150
	attribute2 (name)	setName	116.84
		delete	-300

TABLE I: Q-Table contents for err2 in Fig. 2

find the best possible sequence of concrete actions to repair each error, where the "best" sequence is the one aligned with the user preferences.

To do so, the Q-learning algorithm is executed during a number of episodes, each episode being a complete iteration that repairs the model. The algorithm starts with the Q-table initialized with zeros, therefore, it attempts to repair the first error found in the model with different actions following a try-and-fail approach. That is, it attempts to repair the error in all its contexts by applying all possible actions (the ones remaining after filtering by context and by repairing). When the repairing succeeds, if the action is aligned with the user preferences it will get rewarded and otherwise punished (negative reward when punishing, positive when rewarding). By using the equation detailed in Section II, the algorithm calculates a weight (Q-value) using these rewards and stores it in the Q-table, indicating how good an action is for repairing an error in a context.

The algorithm continues trying different actions for each context in the error, calculating and storing its weights in the Q-table until there are no more errors to repair in the model and the episode finishes. When facing a combination of errors, contexts and actions, with an already calculated weight, the weight gets updated; the more a weight gets updated the more precise it will become. For instance, if the algorithm learns the same action in a context is able to repair an error multiple times, the weight will be higher, reflecting the quality of that action. Eventually, the weight will be so high that the algorithm learns that combination of action and context is the best repairing for the error. Table I shows the Q-table contents for err2 in Fig. 2 after several repairing episodes, with weights calculated by following the preferences of a user who prefer not to delete elements in the model and to repair errors as high in the context hierarchy as possible.

Once each episode finishes, the algorithm generates a sequence with all concrete repairing actions found during that episode, which is stored for later evaluation. Thanks to the explorative (random) nature of RL, episodes generate different sequences. When all episodes finish, the algorithm evaluates all sequences found. To do so, it calculates a total weight for each sequence by adding the weights of the contained actions. Finally, the algorithm compares the total weight of all sequences, selecting the one with highest value. All actions contained in the selected sequence get their Q-table weights increased so that the algorithm learns which actions led to repairing the model. For the example in Fig. 2, the algorithm repairs class Person by choosing a sequence of actions (see Repairing sequence) which comply with the aforementioned simulated preferences: not to delete elements in the model and

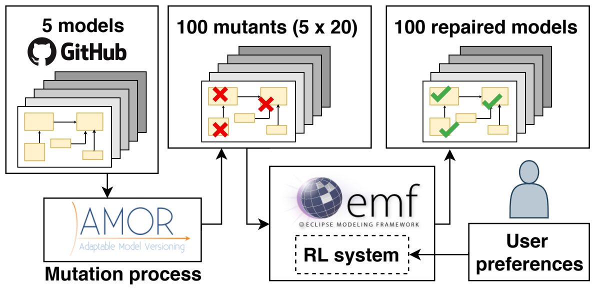

Fig. 3: RL model repair evaluation using mutation testing

to repair errors as high in the context hierarchy as possible (first attribute name is renamed to age).

IV. Evaluation

In this section, we perform an evaluation of our approach by repairing a set of 100 broken models using RL. We start by introducing the evaluation setup and data preparation, followed by an example of how one concrete model is repaired. Finally we present the obtained results. The objectives of this evaluation are to prove that RL can (i) repair a set of real-world models, (ii) learn while repairing them (improving its performance with time), and (iii) perform repairing while respecting user preferences. The evaluation source code is available in [12]. Figure 3 shows an overview of the evaluation. We implemented a RL system to repair broken models using Java in Eclipse Oxygen (the Modeling package). The system code is executed on a laptop with the following specifications: Windows 10 Home, Intel Core i5-6300U @2.4GHz, 64 bits, 16GB RAM. To evaluate our approach, we have selected 5 Ecore [11] models extracted from GitHub. We selected them to show that the algorithm is able to deal with real-world models. The models are taken from the following projects: RandomEMF [13], OCCIware ecore [14] (from which the model in Section IV-A is extracted), amlMetaModel [15], EMF-fragments [16] and MDEForge [17]. Although they can also be considered as metamodels, we treat them as models since we search and repair well-formedness errors with respect to the Ecore metamodel [11].

To obtain a set of broken models for our experiment, for each of these Ecore models we created 20 mutant models by using AMOR Ecore Mutator [18], an EMF-based framework to randomly mutate models conforming to the Ecore metamodel. Each mutant is a modification of an original model introducing inconsistencies (see Fig. 4 in Section IV-A for a sample mutant). The criteria for choosing the 5 original models was that (i) AMOR could process them and that (ii) the introduced errors had enough diversity; we observed that errors were almost the same for every mutant of some of the models. If the produced mutants did not match this criteria, its original Ecore file was discarded and we proceeded with another one. In our preparation, we created a total of 100 mutant, broken models. These mutants have different sizes, from small models (3 classes, less than 10 references, attributes and operations, size of 5KB) to big ones (85 classes, +1400 references, 400 attributes, 5000 operations, size of

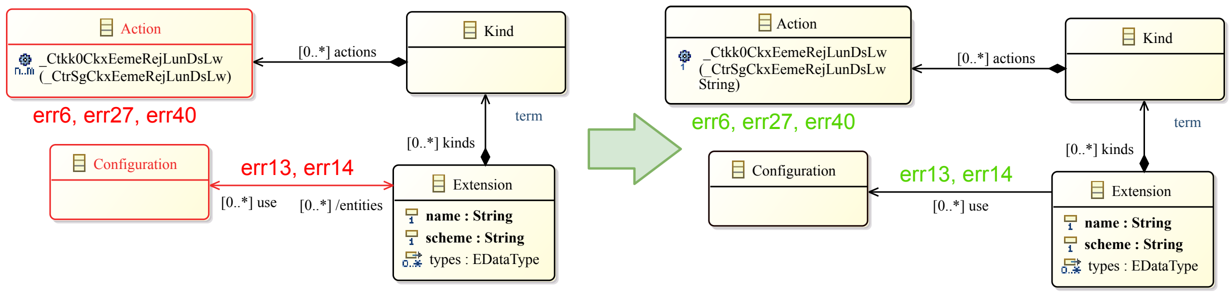

Fig. 4: Example mutant from OCCI ecore before and after repairing

280KB). AMOR is able to introduce 6 types of mutations: add annotations, add objects, delete objects, move objects, unset features and update features. All of these mutations are used in this evaluation. Moreover, 16 different types of errors were reported in the mutants. The number of errors introduced in each model ranged from 1 to 13.

For this evaluation, we have simulated the following personalization preferences: (i) avoid deletion of elements from the model, (ii) repair errors as high in the error context hierarchy as possible and (iii) punish modification of the original model structure. Preferences (i) and (ii) are the same as the ones applied in the example in Fig. 2.

A. Example

Next, we show an example of how one of the mutant models is repaired. The selected mutant originally contains 15 classes, 30 references, 25 attributes, 1 operation and 4 datatypes. To preserve the length of this paper, Fig. 4 shows only faulty parts of the model.

First, the EMF diagnostician extracts all errors present in the model. For this model, it finds 5 errors, their locations are highlighted in Fig. 4 with their respective error codes (see details in Table II). Regarding err6, when defining operations in EMF, modelers can set an upper and lower bound [11] for the returned attribute. Such bounds can be used to make the value returned by the operation a multi-valued attribute. This is, containing distinct values, e.g., if we had an attribute called hobby of type string with a lower bound of 2 and upper of 3, this would mean the attribute would contain a list with at least 2 strings and a maximum of 3. By default, operations have a lower bound of 0 and an upper of 1, since their return type is a singled-valued type. Also, Action class in Fig. 4 contains a method named with a sequence of random characters (the same name is used for the parameter inside the method). These names are part of the mutations introduced by AMOR, but since they do not contain any characters unsupported by EMF, introducing them did not produce any errors.

Then, after filtering actions available in EMF by context and repairing (as detailed in Section III), 11 different actions remain (the same action might be possible to apply in different errors). Next, the Q-Learning algorithm runs during 15 episodes (according to our testing, between 10 and 20 episodes are enough for the algorithm to converge), each of them with a maximum of 20 steps (an episode will finish once the maximum steps value is reached or when there are no more errors to repair in the model, 20 steps provide the algorithm enough time but not too much to avoid falling in never-ending repairing loops). In each step, one action is selected, whether from the Q-Table (the one with highest weight) or randomly. When each episode ends, the algorithm returns a sequence of actions.

Error Code	Message
err6	The lower bound X must be less or equal to the upper bound Y
err13	The opposite must be a feature of the reference's type
err14	The opposite of the opposite may not be a reference different from this one
err27	An operation with void return type must have an upper bound of 1 not X
err40	The typed element must have a type

TABLE II: EMF diagnostician's details of errors in Fig. 4

Finally, when all episodes finish, the algorithm picks the best sequence found (discarding those that do not match the user preferences) and exports the repaired model. Figure 5 shows all sequences found for repairing the model in Fig. 4. For better understanding, we include parameters used in each repairing action. At the moment, these parameters are selected automatically by the algorithm and they work as placeholders. In the future, users will teach the algorithm which parameters they prefer. It is necessary to clarify that err13 gets repaired when err14 does, that is why it does not appear in any of the sequences. Also, err14 appears twice in sequences 4 and 5 because, when deleting part of the reference, the error reappears on its opposite. Sequences from 2 to 5 are discarded since they delete elements in the model and therefore do not match the user preferences. Finally, sequence 6 is the one chosen for repairing the mutant model. It repairs err6 twice, since after setting err27's upper bound to 1, the lower bound is higher thus leading to a new err6. Despite this, sequence 6 has a higher total weight than sequence 1, as for preserving the original model structure, sequence 1 implies changing the return type of an operation to repair err27 and gets punished.

B. Results

This subsection discusses the results obtained from the evaluation. A set of 100 mutant models are repaired, repairing a total of 339 errors among all of them. All mutants follow the repairing preferences presented in Section IV-A: (i) avoid

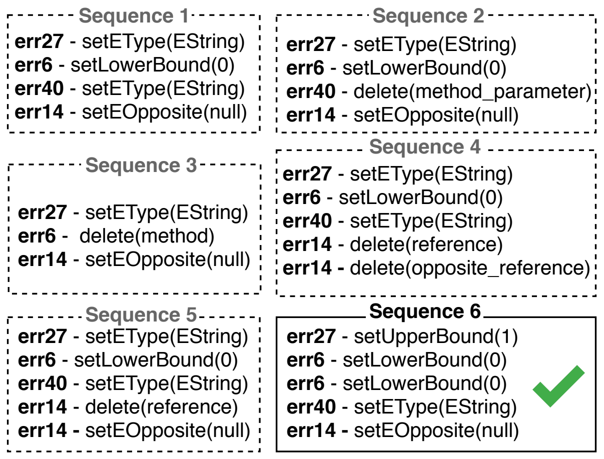
Sequence 1
err27 - setEType(EString)
err6 - setLowerBound(0)
err40 - setEType(EString)
err14 - setEOpposite(null)
Sequence 2
err27 - setEType(EString)
err6 - setLowerBound(0)
err40 - delete(method_parameter)
err14 - setEOpposite(null)
Sequence 3
err27 - setEType(EString)
err6 - delete(method)
err14 - setEOpposite(null)
Sequence 4
err27 - setEType(EString)
err6 - setLowerBound(0)
err40 - setEType(EString)
err14 - delete(reference)
err14 - delete(opposite_reference)
Sequence 5
err27 - setEType(EString)
err6 - setLowerBound(0)
err40 - setEType(EString)
err14 - delete(reference)
err14 - setEOpposite(null)
Sequence 6
err27 - setUpperBound(1)
err6 - setLowerBound(0)
err6 - setLowerBound(0)
err40 - setEType(EString)
err14 - setEOpposite(null)
✓

Fig. 5: Sequences found for repairing the example mutant

deletion of elements from the model, (ii) repair errors as high in the error context hierarchy as possible and (iii) punish modification of the original model structure. For half of the models, one repairing sequence is found, for the rest we are able to find between 2 and 5 possible solutions per model. Sequences not matching user preferences are not included (for example, of the sequences shown in Fig. 5, only 1 and 6 would be included, since the other 4 do not match the requirement of not deleting elements in the model). As stated in Section IV, there are 20 mutants per original model, a total of 5 batches of mutants. The order of repairing is one batch after another (from now on batches A to E), it takes 48.8s (48842ms) to repair all mutants in this order: (a) RandomEMF (models 0-19), (b) OCCIware ecore (models 20-39), (c) amlMetaModel (models 40-59), (d) EMF-fragments (models 60-79) and (e) MDEForge (models 80-99).

Figure 6 shows how much time (in miliseconds) it takes to repair each model (axis X represents model's number). It is worth appreciating how the execution time drops drastically when working within the same batch of mutants (mutants from same original model tend to have similar errors). Execution time increases when facing a new batch and therefore new errors. Also, the initial peak of batch A (model 0) is bigger than peaks in B, C and D (20, 40 and 60) since they share many errors, this is especially visible in B. However, batch E's peak is higher than A's, due to its size (batch E is the one with biggest models). Also, although execution time tends to improve in batch E, it is more unstable than other batches due to these models size and diversity of errors.

To provide further testing on how performance time improved when facing the same errors, we repaired all mutants three consecutive times. Repairing the 100 mutants took 48.8s, 33.4s and 28.3s, respectively in each round. Performance gets faster each time, especially from the first to the second repairing round, where repairing time got reduced by a 31,48% while from the second to the third one there was a 10,47% improvement.

In conclusion, all mutant models in the evaluation are repaired accordingly to the user preferences. After execution, a

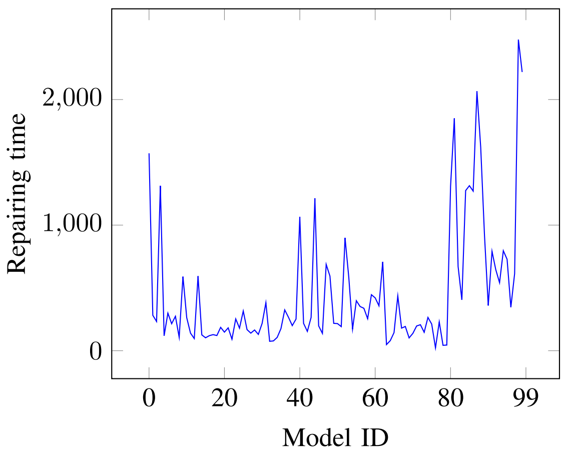

Fig. 6: Evolution of execution time in ms for each repairing

repaired Ecore file for each of them is exported. For half of the models the algorithm was able to find more than one solution. Our approach is able to learn from errors already repaired and therefore repair them when faced again in a more efficient way. We consider these results successful and we believe further research in this direction could be beneficial for the model repair field.

V. Threats to validity

Although we consider the evaluation successful in proving that RL can be a good solution for personalized and automatic repair of models, we face some validations issues that are worth discussing in this section.

The reasons for using AMOR are: its easy integration with EMF and the randomness of the introduced mutations. Other tools were also pondered, like Wodel: a Domain-Specific Language for the specification and generation of model mutants [19]. However, it was discarded since it requires to define which specific mutations will be introduced, and we did not want to be in control of the produced mutants in order to enhance the validity of the evaluation.

The 5 original Ecore files used in the evaluation were selected arbitrarily from GitHub, being the only requisite that AMOR could process and introduce errors with diversity. We opted to search for files in GitHub because we could not find any benchmark for testing model repairing tools. These 5 files are different from one another, but they do not cover all possibilities within Ecore models. We also considered to use an instance generator to create Ecore models ramdomly and then mutate them. This possibility was discarded because generator tools create synthetic models and we wanted the mutants to be as close as possible to real models.

Additionally, mutants from the same model have similar errors between them. This is beneficial for the evaluation as it shows how the algorithm actually learns from already repaired errors, but it also reduces the variety of errors tested. Another issue is that, despite the random nature of AMOR, it has a predefined set of mutations, and the errors it produces might not be as complex as errors introduced by a human.

Regarding personalization, no external user participated during the evaluation. Instead, we simulated some user preferences and implemented them into the reward system. Although these preferences were fictitious, we believe they were close enough to what a real modeler could have selected.

The source code of the implemented evaluation together with the mutant models we repaired are allocated in [12], available to download and test. The information provided in Sections III and IV shows transparency about our research, which should make the process explained in this paper replicable.

VI. Related work

Model repair is a research field which has drawn the interest of many researchers to formulate approaches and build different tools to repair broken models. The main feature that distinguish our approach from others is the capability to learn from each model repaired in order to streamline the performance.

We could not find in the literature any research applying RL to model repair. The most similar work to ours we could find is [20], where Puissant et al. present Badger, a tool based on an artificial intelligence technique called automated planning. Badger generates sequences that lead from an initial state to a defined goal. It has a set of repairing 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 sequences to repair the whole model, since some repair actions can modify the model drastically, and we consider counter-intuitive to decide which action to apply without knowing its consequences. Additionally, RL performs better after each execution, while automated planning does not.

Nassar et al. [21] propose a rule-based prototype where EMF models are automatically completed, with user intervention in the process. Our approach has more autonomy, since user preferences are only introduced at the beginning of the repair process and not during. Rule-based approaches excel when repairing the same errors in the same scenario. Our approach repairs not only errors already faced but also similar ones, in the same or different scenarios. In the evaluation presented in Section IV, we prove our approach is able to repair errors of different nature found in different models.

Taentzer et al. [22] present a prototype based on graph transformation theory for change-preserving model repair. In their approach, authors check operations performed on a model to identify which 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 [23] 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 fixes for a given problem.

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 showed promising results dealing with model transformations and evolution scenarios, for example in [24] 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.

VII. Conclusions and future work

In this paper, we presented an approach to repair models using RL. With RL we are able to repair models automatically and to provide personalized results. By taking into account user preferences, RL rewards can be modified to produce different repairing sequences. We presented an evaluation of our approach, in which we repair a set of broken mutant models. To show how the algorithm provides personalized repairs, we simulated some user preferences such as avoiding deletion of model elements. The evaluation showed how Q-Learning repaired and provided personalized solutions for all our 100 test models. This works as a proof of concept and shows the benefits of repairing models using RL. Our results are promising and can be seen as an indicator of the potential of this research direction.

In the future, we would like to work further on personalized repairing, first by providing users a mean to introduce their own preferences, through an interface or domain specific language. We are aware of the limitations of fully automated model repair, hence we plan to research which are the best terms to keep a human in the loop while repairing. Also, we want to focus on how users can improve the results obtained with RL when facing situations where different actions are able to repair the same error. It is essential to find the balance between personalized results and a generic approach, where knowledge gained from repairing models under certain preferences can be reused when working with new broken models and different preferences. The focus will be to research how to store this knowledge so that personalization from one user do not prejudice other users experience.

At the moment, our implementation only repairs models with respect to the Ecore metamodel, future development will include support for domain-specific metamodels.

So far, we repair errors in the order they are returned by EMF, we are aware altering this order will have consequences in the produced sequences and therefore we need to perform further testing in this direction. Now, knowing that RL can be used to repair models, we would like to put more focus on assuring the quality of repaired models. So far, the only measurable quality is how much it fits user preferences. In the future, we want not only to produce correct models but also to enhance their quality based on metrics [25]. One exciting area we want to study is the refactoring of models using RL to make them more aligned to architectural patterns given by textbooks or companies. Additional rewards could be related to how well the models meet the coupling and cohesion criteria.

Finally, we would like to test other scenarios and modeling environments for model repairing, as well as other RL algorithms [4].

References

[1] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2014.

[2] N. Macedo, T. M. S. Jorge, and A. Cunha, "A feature-based classification of model repair approaches," 2017.

[3] A. Barriga, A. Rutle, and R. Heldal, "Automatic model repair using reinforcement learning," in *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018.*, 2018, pp. 781–786. [Online]. Available: http://ceur-ws.org/Vol-2245/ammore_paper_1.pdf

[4] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," 2011.

[5] J. Cabot, R. Clarisó, M. Brambilla, and S. Gérard, "Cognifying model-driven software engineering," in *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 2017, pp. 154–160.

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

[7] B. Karasneh and M. R. Chaudron, "Online img2uml repository: An online repository for UML," in *EESSMOD@ MoDELS*, 2013, pp. 61–66.

[8] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, "Mdeforge: an extensible web-based modeling platform." in *CloudMDE@ MoDELS*, 2014, pp. 66–75.

[9] R. Bellman, *Dynamic programming*. Courier Corporation, 2013.

[10] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[11] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[12] A. Barriga, "Project PARMOREL." [Online]. Available: https://ict.hvl.no/project-parmorel/

[13] markus1978, "markus1978/randomemf," Dec 2015. [Online]. Available: https://github.com/markus1978/RandomEMF/

[14] Occiware, "occiware/ecore," Sep 2017. [Online]. Available: https://github.com/occiware/ecore/

[15] amlModeling, "amlmodeling/amlmetamodel," Jan 2016. [Online]. Available: https://github.com/amlModeling/amlMetaModel

[16] markus1978, "markus1978/emf-fragments," May 2014. [Online]. Available: https://github.com/markus1978/emf-fragments

[17] MDEGroup, "Mdegroup/mdeforge," Mar 2018. [Online]. Available: https://github.com/MDEGroup/MDEForge

[18] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer, "Amor–towards adaptable model versioning," in *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*, vol. 8, 2008, pp. 4–50.

[19] P. Gómez-Abajo, E. Guerra, and J. de Lara, "Wodel: A domain-specific language for model mutation," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. New York, NY, USA: ACM, 2016, pp. 1968–1973. [Online]. Available: http://doi.acm.org/10.1145/2851613.2851751

[20] J. P. Puissant, R. Van Der Straeten, and T. Mens, "Resolving model inconsistencies using automated regression planning," *Software & Systems Modeling*, vol. 14, no. 1, pp. 461–481, 2015.

[21] N. Nassar, H. Radke, and T. Arendt, "Rule-based repair of EMF models: An automated interactive approach," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2017, pp. 171–181.

[22] G. Taentzer, M. Ohrndorf, Y. Lamo, and A. Rutle, "Change-preserving model repair," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2017, pp. 283–299.

[23] R. Kretschmer, D. E. Khelladi, and A. Egyed, "An automated and instant discovery of concrete repairs for model inconsistencies," in *Proceedings of the 40th ICSE: Companion Proceeedings*. ACM, 2018, pp. 298–299.

[24] M. Kessentini, U. Mansoor, M. Wimmer, A. Ouni, and K. Deb, "Search-based detection of model level changes," *Empirical Software Engineering*, vol. 22, no. 2, pp. 670–715, 2017.

[25] T. Arendt, P. Stepien, and G. Taentzer, "EMF metrics: Specification and calculation of model metrics within the eclipse modeling framework," in *of the BENEVOL workshop*, 2010.