Automatic model repair using reinforcement learning

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

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

Rogardt Heldal
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. One option to prevent and repair broken models is to automatize this process with a software tool, using techniques like Machine Learning (ML). Despite its potential to help fixing models, it is still challenging to apply ML techniques in the modeling domain due to the lack of available datasets. However, some ML branches could offer promising results since they do not require initial training data, such as Unsupervised and Reinforcement Learning (RL). In this paper we present a prototype tool for automatic model repairing by using RL algorithms. These algorithms could potentially reach model repairing with human-quality without requiring supervision. To conduct an initial evaluation we have tested our prototype with a set of broken models and studied its execution and results.

CCS CONCEPTS

• **Computing methodologies** → **Reinforcement learning**; **Model verification and validation**; • **Software and its engineering** → *Risk management*; • **Theory of computation** → Theory and algorithms for application domains;

KEYWORDS

Model Repair, Machine Learning, Reinforcement Learning

1 INTRODUCTION

Models are used to develop key parts of systems in engineering domains [23]. Therefore, the correctness and accuracy of such models are of the utmost importance to maintain good quality during the development of these systems. The complexity of keeping models free of errors grows together with the size of working teams and number changes during the development process. Tracking all versions of the same model and checking its correctness can be a challenging task, especially if also the size of the model increases. Automation can be a good solution to ease the complexity of this process by periodically checking if a model is free of errors and repairing them when they occur. Tools that automatize or support error detection and repairing of models can greatly improve how organizations deal with Model-Driven engineering processes by reducing the burden of manually dealing with correctness issues and improving delivery time and final quality.

To this end, there already exists many tools and research efforts [16] for automatically resolving software bugs that have shown promising results. Examples of developed prototypes for model repairing can be found in [6] where authors achieve repairing by aligning models and their behavior logs or in [17] where EMF models are repaired by using a rule-based interactive approach. Other contributions take into account the editing history of a model such as [18] with a tool to generate repair suggestions and [20] with a prototype based on graph transformation theory.

However, and despite the potential already shown by Machine Learning (ML) in achieving (and sometimes even surpassing) human performance in repetitive tasks, we could not find in the literature any research applying ML to improve how developers deal with cyclical tasks in modeling, such as repairing. Authors in [3] stress how the combination of these two fields, ML and modeling, could be really beneficial for the future of modeling and state some application examples. ML has already shown its effectiveness providing support and independently solving different tasks in software engineering like testing [5], quality assurance [4, 15] or verification [8].

The biggest challenge for ML adoption within modeling is the lack of historical data available publicly, since most ML algorithms need great amounts of datasets in order to achieve high-quality results. Some available model repositories are [1, 7, 11], yet the data they offer is limited in terms of diversity and labeling. Still, until the available model datasets grow and improve, there are some ML algorithms that could work overcoming this data issue. Unlike other ML techniques, Reinforcement Learning (RL) does not require training datasets, since their purpose is to find structures in the data they work with without supplying any target or outcome beforehand. This make them useful in situations and domains which lack historical data.

Another issue is that the size of search space when performing automatic repairing can grow exponentially when the target model has enough content. In addition, potential interesting fixes are usually not unique, the same model could be repaired in many different ways, adding complexity and uncertainty into ML algorithms.

RL offers algorithms able to learn by themselves how to interact in an environment by providing them a set of possible actions and rewards for each of them. Using these rewards they can learn which are the best actions to perform in said environment. Rewards can be established so that algorithms learn how to fix undesired situations or problems in the environment, such as conflicts with non-unique repairing actions. The search space can be reduced by classifying actions available accordingly to the error to fix. Actions that could never interact with an error could be omitted.

This technique dates from the 80s, however, it has been during last years due to the decreasing price of hardware and increasing of computational power, that it has produced breakthrough results in tasks as different as: reaching above-human-level performance in video and board games [19], improving existing algorithms [14] or teaching simulation entities and robots themselves to walk [9].

Given the potential these algorithms offer together with the need of automated tools for model repair, we consider that studying how to apply RL for autonomous model repair could provide interesting results, with the possibility to achieve human performance in fixing models.

Hence in this paper we present a prototype of how a RL algorithm can repair errors in a set of broken models without human intervention, showing its execution and results. In this first approach, we consider errors related to violations of well-formedness constraints, with the idea to generalize the type of errors supported after further development.

2 AUTOMATIC MODEL REPAIR TOOL

The Automatic Model Repair Tool (AMRT) is able, by using RL techniques, to choose and apply the best actions to fix a broken model. During this process the tool is independent from the user and does not require any supervision.

2.1 Algorithm

RL consists of algorithms that learn what to do, given a situation and a set of possible actions to choose from, in order to maximize a reward. These algorithms include an agent that must learn by itself what to do through interacting with the environment. Learning is accomplished by choosing those available actions with best rewards.

In the model repair scenario, the agent is the model to be repaired and the environment the framework where it was defined (EMF in our approach). EMF provides error diagnostics about the model, which can be altered by actions available in the framework. If the algorithm selects actions reducing the number of errors in the model it gets a positive reward, contrarily it is penalized. This algorithm workflow is detailed in Fig. 1.

When there are no more errors to solve the environment returns a maximum reward, the ultimate goal to achieve. This guarantees to solve all errors on the model but since there might be many ways to solve them, the algorithm might not find the optimal solution. However, it could be reached by providing the algorithm with more models to solve so that it learns further or by manually favoring some rewards over others.

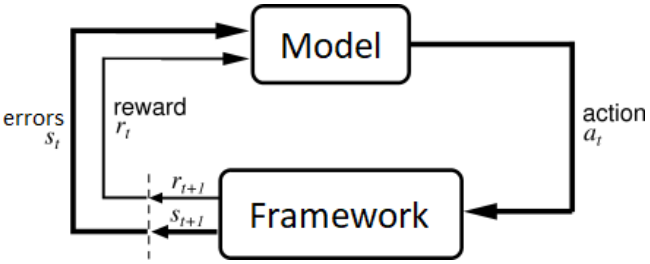

Figure 1: Reinforcement learning applied to model repair workflow

AMRT is powered by a Q-Learning algorithm [22]. We will apply this algorithm to a sample EMF model in section 2.3, here we only explain the main characteristics of the algorithm.

In Q-Learning, the agent chooses the most optimal action by consulting a table structure called Q-table. This table is updated while the agent interacts with the environment with repeated calculations of the Bellman Equation [2]. This equation returns a Q-value telling that the maximum future reward is the reward (r) the agent received for entering the current state (s) with some action (a) plus the maximum future reward for the next state (s') and action (a') reduced by a discount factor (gamma) to avoid falling in a local maximum (see Fig. 1). This allow to infer the value of the current state (s) based on the calculation of the next one (s'), which can be used to calculate an optimal policy to select actions.

$$Q(s,a) = r + \gamma(\max_{a'}(Q(s',a'))$$

From here the algorithm can determine the utility of a state and the optimal action to do next until it reaches its final goal (maximum reward). Each iteration (attempt made by the agent) is called an episode. For each episode, the agent will try to achieve the goal state and for every transition (pair of state and action) it will keep on updating the values of the Q table.

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

2.2 Phases

The tool undergoes two different phases: learning and running. During the learning phase, the Q-learning algorithm will be trained using one or different models, updating its Q-table and improving the general Q-value. In other words, learning which are the optimal actions to fix a broken model.

The algorithm will receive as initial input:

(1) An agent: model in XML format.
(2) Actions: given Ecore and Ecore-editor, find the exhaustive set of editions which are possible actions available to perform on the model.
(3) State: given a model, get an error diagnostics from the EMF tool.

As the algorithm is executed, for each action performed on the model it will need to know its current state, namely it will need to be connected to EMF to iteratively update the model and analyse its errors. The Q-value will be optimized until the model has no errors left. This structure can be seen in detail in Fig. 2 and as pseudocde in Alg. 1. The more models provided to the algorithm the better performance it can get.

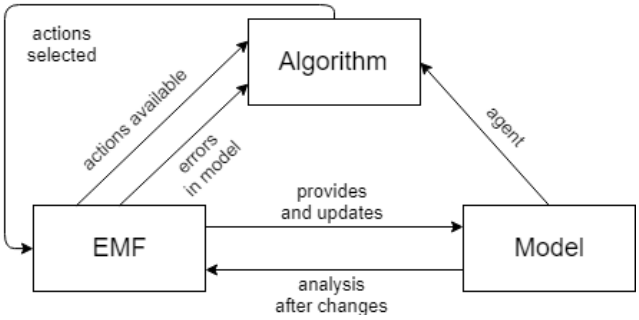

Figure 2: Tool training structure

Once the algorithm has improved enough, the tool can enter into the running phase like a trained package pluggable to EMF. As a plugin it will be able to analyze and repair models directly on EMF without any supervision required. Still, it could be possible to open its behavior for human interaction, allowing users to give feedback to the tool about decisions made so that it can learn from human choices and to personalize its performance.

2.3 Example

In this section, we introduce a running example in AMRT and demonstrate how the algorithm repairs all errors in a sample model from scratch. For this example we have assumed the available actions to perform are:

(1) Action 0 - Rename class
(2) Action 1 - Delete super type
(3) Action 2 - Rename attribute
(4) Action 3 - Add type to attribute

Algorithm 1 AMRT Algorithm

```
Initialize: Q-matrix, Gamma
INPUT: from EMF (Model, Actions, Errors)
while Episodes not empty do
    for each Error s do
        for Steps in Episode do
            Select Action a // random or highest Q-value
            Evaluate changes in Model produced by a
            Update Q-table
        if s is solved then
            Delete s
            Exit for
        else
            s' ← s // s' stores s modified by a
            Save s' in Errors
```

We have preferred to omit delete class/attribute actions, keeping the example as simple as possible since they could also technically solve errors in the models but would not provide correct solutions (e.g.; by deleting the class Web its attributes would have no duplicated names anymore). However, this behavior can be controlled by reducing the rewards of this kind of actions and they would not be a problem in a real environment.

The algorithm counts with a maximum of 20 episodes and 3 steps per episode, this number is low as it also is the amount of pairs $(action, error)$ available, allowing us to take further insight about how the algorithm interacts with the model (contrarily it would find the solution faster but its decision process would not be so visible).

Additionally, the algorithm can sometime pick random actions instead of the one with biggest reward, in order to find new and perhaps more optimal solutions. Our random probability for this example is 10% since it is a really controlled environment and our objective is to prove how the algorithm learns by itself.

During this section errors will be identified using numbers:

(1) Error 0 - Duplicated attribute name.
(2) Error 1 - Attribute with missing type.
(3) Error 2 - Class super type of itself.
(4) Error 3 - Duplicated class name.

The selected model represents part of a Web structure and consists of three classes related with a reference, a composition and a super type link, as can be seen in Fig 3.

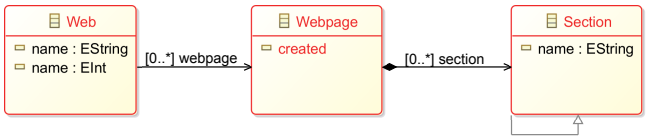

Figure 3: First sample model

Each class has an error:

(1) Web: duplicated attribute name (Error 0).
(2) Webpage: attribute without type (Error 1).
(3) Section: the class is super type of itself (Error 2).

Table 1 details the execution of the algorithm on the sample model step by step. The correct form to read the table is, for each episode, to read steps from left to right consecutively (if there are two rows of steps in the same episode the correct way would be to read all steps on the first one and then the second). Cells in red indicate that the selected action does not solve the current error, while green implies solving the error. (R) after an action means it was selected randomly.

Table 1: Algorithm execution on sample model

Episodes	Step 1	Step 2	Step 3
1	Error 0 <- Action 0	Error 0 <- Action 2 (R)	
	Error 2 <- Action 0	Error 2 <- Action 1	
2	Error 1 <- Action 0	Error 1 <- Action 1	Error 1 <- Action 2
3	Error 1 <- Action 1 (R)	Error 1 <- Action 3	

The algorithm starts without any knowledge about the environment, therefore it starts by exploring for each error the different actions available. First attempt is to solve Error 0 using Action 0 without success. The algorithm updates then the Q-table and learns that Action 0 does not solve Error 0. Next action is picked randomly and it solves Error 0. Here the randomness component has increased the speed for finding a solution. Next error to solve is Error 2, since it is the first time the algorithm processes this error, it tries actions available one by one. It finds the solution by applying Action 1. Next the algorithm faces Error 1, again an unknown Error. This time we can see how it explore all actions (even trying a random repetition with Action 1) until it finds the solution in the last one, Action 3.

After a few iterations, the algorithm has been able to fix all errors in the model without any previous training.

2.4 Evaluation

Once the algorithm has learned about the environment, it is interesting to evaluate its flexibility by repairing other models and how its performance improves.

We have applied the AMRT on a set of models created with errors reparable by the available actions stated in Sec. 2.3.

The set contains four models. For each model we include an enumeration of present errors, its diagram and a table showing how its is fixed by the algorithm. We present them in the order they are processed by the algorithm. For each model the Q-table is updated and saved for the next one.

2.4.1 Second model.

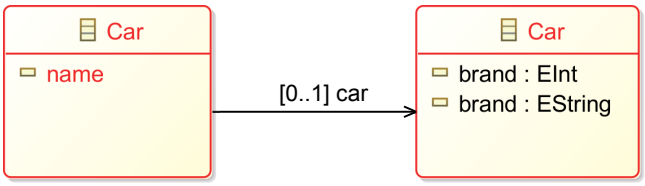

Figure 4: Second sample model

Errors:

(1) Car: duplicated class name (Error 3).
(2) Car: attribute without type (Error 1).
(3) Car': duplicated attribute name (Error 0).

Table 2: Algorithm execution on the second model

Episodes	Step 1	Step 2
1	Error 3 <- Action 1 (R)	Error 3 <- Action 0
	Error 0 <- Action 1	Error 0 <- Action 2
2	Error 1 <- Action 3	

With this model, a new error is introduced into the system, Error 3. One can see in Table 4 that the algorithm starts by selecting for it a wrong random action but is able to fix it with the second action chosen. Next, although Error 0 is already known, Q-table still needs to be tuned and it finds a solution on step 2. Finally, it is able to find a solution for Error 1 at the first attempt.

Although it is only the second model processed by the algorithm, the fixing process is already performed faster.

2.4.2 Third model.

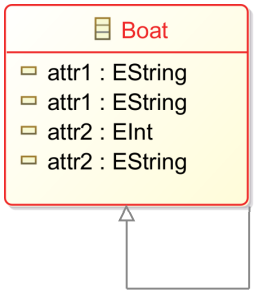

Figure 5: Third sample model

Errors:

(1) Boat: duplicated attribute name, twice (Error 0).
(2) Boat: the class is super type of itself (Error 2).

Table 3: Algorithm execution on the third model

Episodes	Step 1
1	Error 2 <- Action 1
	Error 1 <- Action 3
2	Error 0 <- Action 2

The algorithm is able to fix this model directly as can be seen in Table 5, already knowing which action solve each error.

2.4.3 Fourth model.

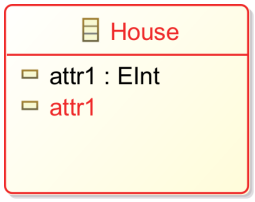

Figure 6: Fourth sample model

Errors:

(1) House: duplicated attribute name (Error 0).
(2) House: attribute without type (Error 1).

Table 4: Algorithm execution on the fourth model

Episodes	Step 1
1	Error 0 <- Action 2
2	Error 1 <- Action 0 (R)
	Error 1 <- Action 2

In execution showed by Table 6, the algorithm selects one wrong action, but only because it is selected randomly.

2.4.4 Fifth model.

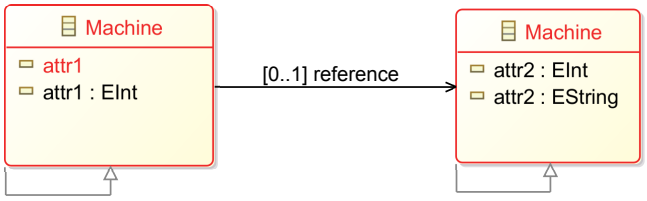

Figure 7: Fifth sample model

Errors:

(1) Machine: duplicated class name (Error 3).

(2) Machine: duplicated attribute name (Error 0).
(3) Machine: attribute without type (Error 1).
(4) Machine: the class is super type of itself (Error 2).
(5) Machine': duplicated attribute name (Error 0').
(6) Machine': the class is super type of itself (Error 2').

Table 5: Algorithm execution on the fifth model

Episodes	Step 1	Step 2	Step 3
1	Error 3 <- Action 0	Error 1 <- Action 3	Error 0 <- Action 2
2	Error 2 <- Action 1	Error 2' <- Action 1	
3	Error 0' <- Action 1		

The final model is the biggest of the set with six errors. Table 7 displays how it is fixed by the algorithm straightforwardly.

2.4.5 Evaluation results.

The evaluation shows that AMRT is able to automatically repair a model without any previous training. Once the algorithm gains enough knowledge, it is able to repair new models straightforwardly as long as they are reparable by the set of known actions.

Performance improves really quick, allowing the AMRT to fix models almost directly after a couple of models processed.

Figure 8 shows how the accuracy percentage evolves through the models processed. Improvement is clear, the percentage increases each iteration except for the fourth model. However, as showed in Table 6, the only wrong action picked for this model was caused by the random dimension of the algorithm (10% of the actions are selected randomly instead of picking the best possible one in the Q-table), and without it 100% accuracy would have been reached.

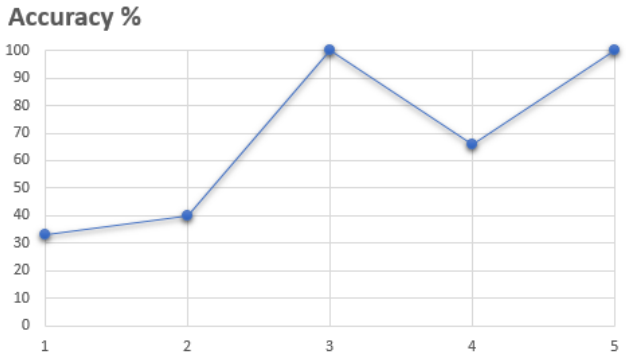

Figure 8: Accuracy evolution of the algorithm

3 RELATED WORK

Model repair is a research field which results can greatly improve how engineers interact with model-driven projects. Thus, it has drawn the interests of many researchers to formulate approaches and build different tools to fix broken models.

Some research tackle model repair relating a given model to its past event logs. Fahland et al. [6] study model repairing by aligning a given model with its behavior logs. Their model repair technique cover the whole state space even if there is total conformance between models and logs. Our proposed technique avoid processing the totality of the state space at once, allowing subdivision for each error and its available actions.

Other efforts focus on rule-based approaches. Nassar et al. [17] present a prototype where EMF models are automatically trimmed and completed, with user intervention in the process. Their tool works as a support guide for modelers. For us, it is also important to provide user interaction but our aim is to give the prototype freedom to perform its own decisions (although they can be supervised by the user if he desires so).

Alternative contributions take into account the editing history of a model such as Ohrndorf et al. [18] with a tool that generates repair suggestions. Their method presents the interesting idea of identifying which change in the editing history originated an error. However, the tool works again just as a mere recommendation system and it does not have decision power. Additionally their approach only works if the model comes together with its edit history, not if the model is introduced with errors from scratch.

Taentzer et al. [20] 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. This could not work in all environments lacking therefore adaptability and reusability, for instance in situations where inconsistencies happen due to simultaneous commits from team members.

Kretschmer et al. introduce in [13] 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. Their approach is similar to ours but we believe RL can reduce state space even more than a tree-based search, especially when tuning rewards and tackling errors independently. Also, 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.

Lastly, again Kretschmer et al. [12] propose an automated tool to repair inconsistencies. They propose generator functions to make their tool more generic and users have to add their own generator functions. We go a step forward using RL techniques that extend the scope of our approach and provide the flexibility for reusing the framework to be applied in a wide variety of model repairing cases. Also, we get the available actions directly from EMF, without needing further intervention from the user.

4 FUTURE WORK

In the following section, we discuss the next research steps and how to improve our current results.

At the moment, the tool simulates to be connected with EMF in order to receive models, errors and actions. Next immediate step is to actually implement and test the tool connected with the framework. Additionally, it will be necessary to develop a plugin and test how it works with models directly on the EMF environment.

In order to reach better results and further validation, we will test the tool on a bigger set of models. These will be obtained by creating a number of EMF models (either manually or from tools like [21], from a public repository such as [1, 7, 11] or by extracting them from general repositories like GitHub by using data mining techniques. These models will be analyzed and repaired by the tool using the complete set of actions available in EMF. We will study how the tool performances in this more complex scenario and modify it until it reaches an optimal performance. Additionally, we will research different modifications of the algorithm in order to find the best way to face model repairing, as well as testing other RL algorithms [10].

Finally, we would like to study further to which degree the interaction between users and the algorithm can benefit the final result, which are the best terms to develop said interaction and to identify the most crucial operations for model developers subject to be improved by using ML. We would like to focus especially on how users can boost the algorithm decidability when facing situations where different actions are able to apply the same fix on the model.

5 CONCLUSIONS

In this paper, we have proposed the use of RL algorithms to fix broken models. Contrarily to other ML techniques, RL algorithms do not need as many data as they do not require an initial dataset and are known for working well in finding solutions in unknown environments.

We present AMRT, a tool that given a broken model and a set of editing actions available, is able to find mistakes and which actions can repair them. This combination of RL and automatic fixing could potentially allow model repairing at human-level without requiring supervision.

While the results obtained are still at an early stage, we consider them promising and an indicator of the potential in this research line.

REFERENCES

[1] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. 2014. MDEForge: an Extensible Web-Based Modeling Platform.. In *CloudMDE@ MoDELS*. 66–75.

[2] Richard Bellman. 2013. *Dynamic programming*. Courier Corporation.

[3] Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard. 2017. Cognifying Model-Driven Software Engineering. In *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 154–160.

[4] Kanika Chandra, Gagan Kapoor, Rashi Kohli, and Archana Gupta. 2016. Improving software quality using machine learning. In *Innovation and Challenges in Cyber Security (ICICCS-INBUSH), 2016 International Conference on*. IEEE, 115–118.

[5] Richard Chang, Sriram Sankaranarayanan, Guofei Jiang, and Franjo Ivancic. 2014. Software testing using machine learning. US Patent 8,924,938.

[6] Dirk Fahland and Wil MP van der Aalst. 2015. Model repair-aligning process models to reality. *Information Systems* 47 (2015), 220–243.

[7] Robert B France, James M Bieman, Sai Pradeep Mandalaparty, Betty HC Cheng, and Adam C Jensen. 2012. Repository for model driven development (ReMoDD). In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 1471–1472.

[8] Oliver Friedrichs, Alfred Huger, and Adam J O'donnell. 2015. Method and apparatus for detecting malicious software through contextual convictions, generic signatures and machine learning techniques. US Patent 9,088,601.

[9] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. 2017. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286* (2017).

[10] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[11] Bilal Karasneh and Michel RV Chaudron. 2013. Online Img2UML Repository: An Online Repository for UML Models.. In *EESSMOD@ MoDELS*. 61–66.

[12] Roland Kretschmer, Djamel Eddine Khelladi, Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. 2017. From Abstract to Concrete Repairs of Model Inconsistencies: an Automated Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 456–465.

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

[14] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*. 6382–6393.

[15] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[16] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 17.

[17] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. 2017. Rule-Based Repair of EMF Models: An Automated Interactive Approach. In *International Conference on Theory and Practice of Model Transformations*. Springer, 171–181.

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

[19] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.

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

[21] Antonio Vallecillo, Frank Hilken, Loli Burgueño, Martin Gogolla, et al. 2016. Generating Effective Test Suites for Model Transformations Using Classifying Terms. (2016).

[22] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

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