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

Model Repair with Quality-Based Reinforcement Learning

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

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

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

Keywords MDE; Machine Learning; Model Repair; Quality Evaluation

1 Introduction

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

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

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

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

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

2 Running example

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

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

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

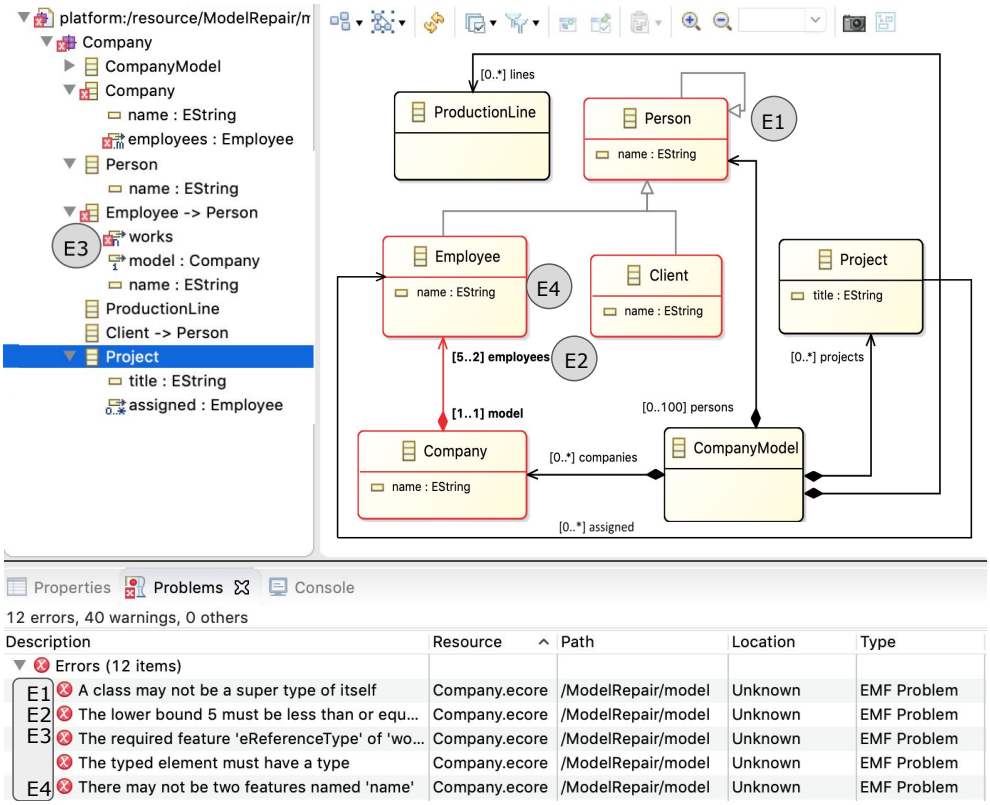

Figure 1 – A snapshot of an invalid domain model

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

E1 Person cannot inherit from itself

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

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

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

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

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

A1 Removal of the supertype relationship

A2 Adding a new class to satisfy the supertyping

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

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

A1 Invert lowerBound with upperBound of the reference

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

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

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

A2 Type it with ProductionLine

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

In this case the resolution can be listed as follows:

A1 Remove the attribute from the supertype

A2 Remove the attribute from the subtypes

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

3 Quality evaluation

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

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

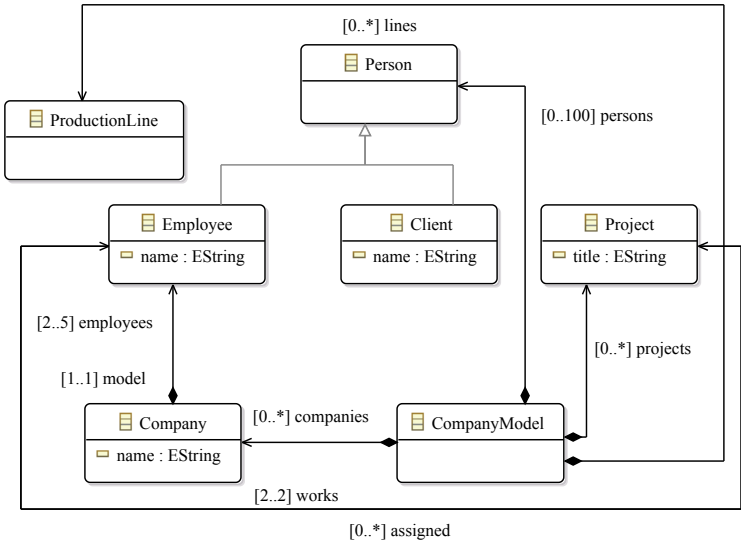

Figure 2 – Repaired model with actions A1

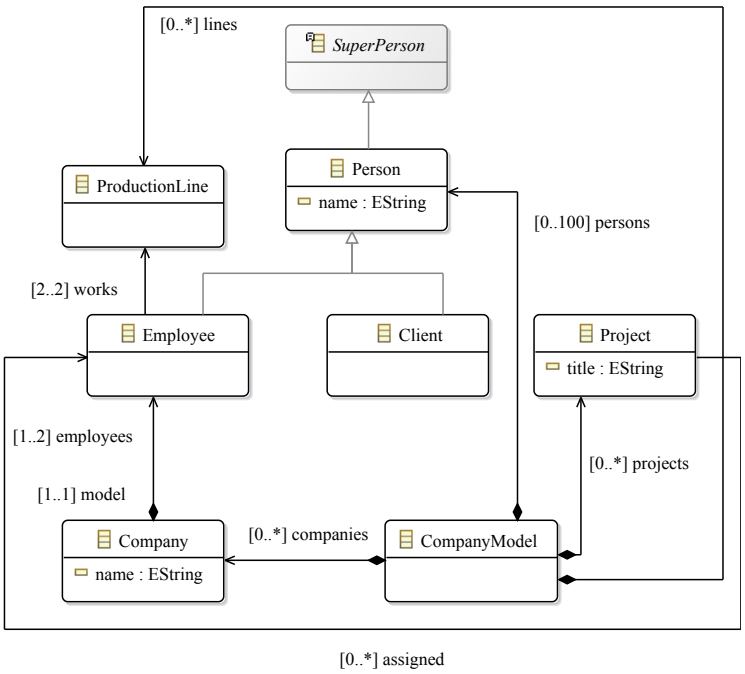

Figure 3 – Repaired model with actions A2

3.1 Quality characteristics

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

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

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

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

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

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

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

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

$$Complexity = (NR - NUR + NOPR + UND + (NR - NCR)) \tag{3}$$

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

Table 1 – Excerpt of the metrics considered in the evaluation

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

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

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

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

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

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

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

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

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

3.2 Evaluating the running example

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

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

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

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

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

4 Customizing quality characteristics with RL

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

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

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

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

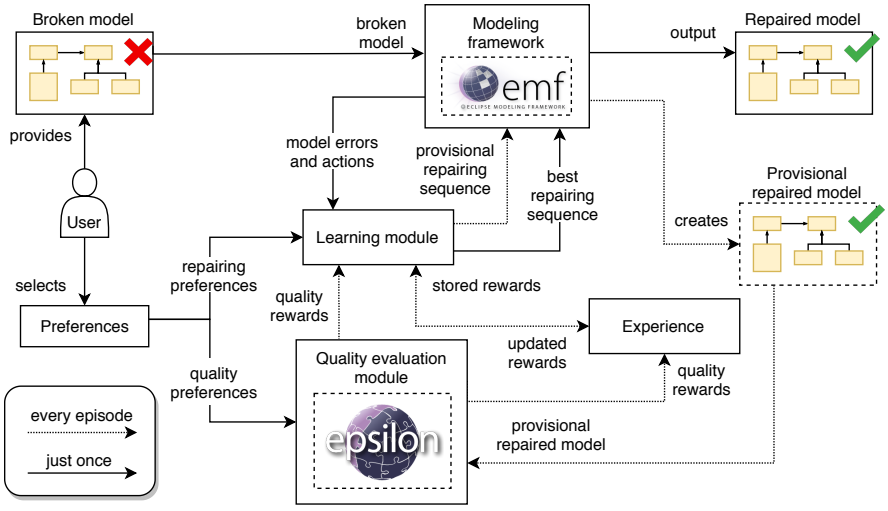

Figure 4 – PARMOREL's workflow for assuring quality in repaired models

4.1 Specification of quality aspects

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

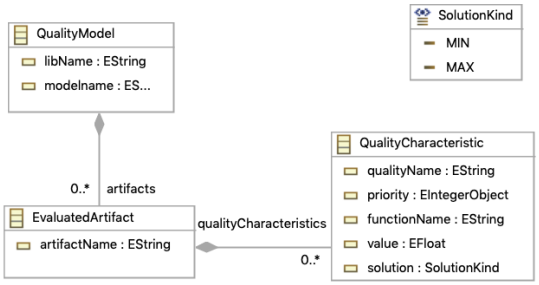

Figure 5 – Quality characteristics model

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

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

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

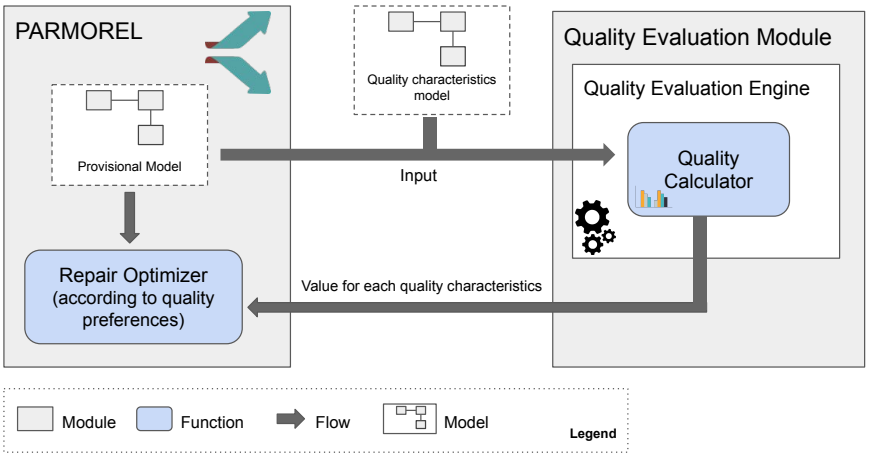

Figure 6 – Excerpt workflow of quality characteristic evaluation

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

Algorithm 1 Quality evaluation, EOL main file excerpt

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

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

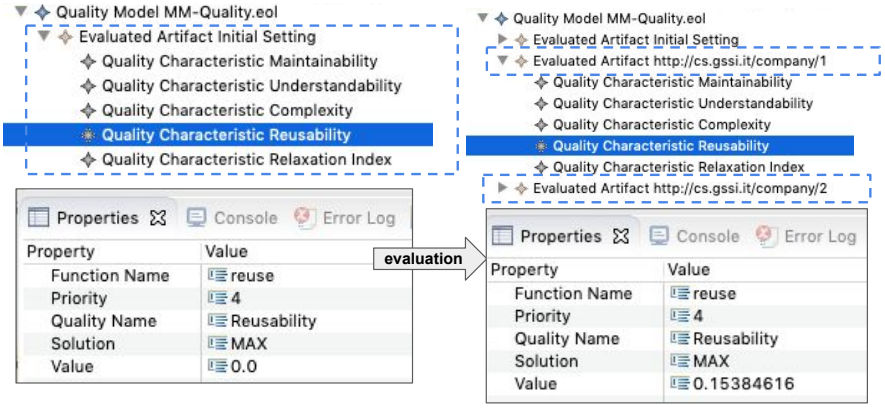

Figure 7 – Quality model initial setting and result

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

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

4.2 Using quality evaluation in preference specification

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

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

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

Algorithm 2 Rewards calculation in PARMOREL

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

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

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

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

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

5 Evaluation of PARMOREL

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

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

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

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

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

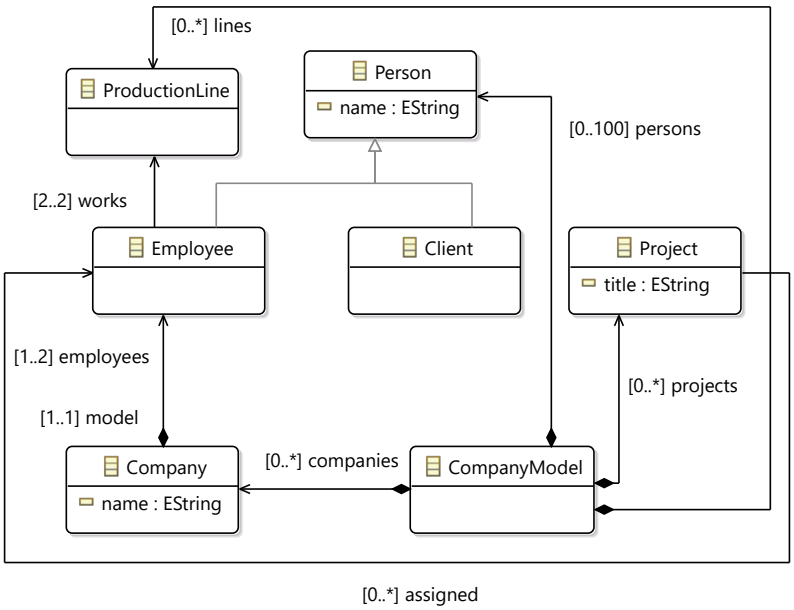

Figure 8 – Model from Fig. 1 prioritizing complexity, reusability and understandability

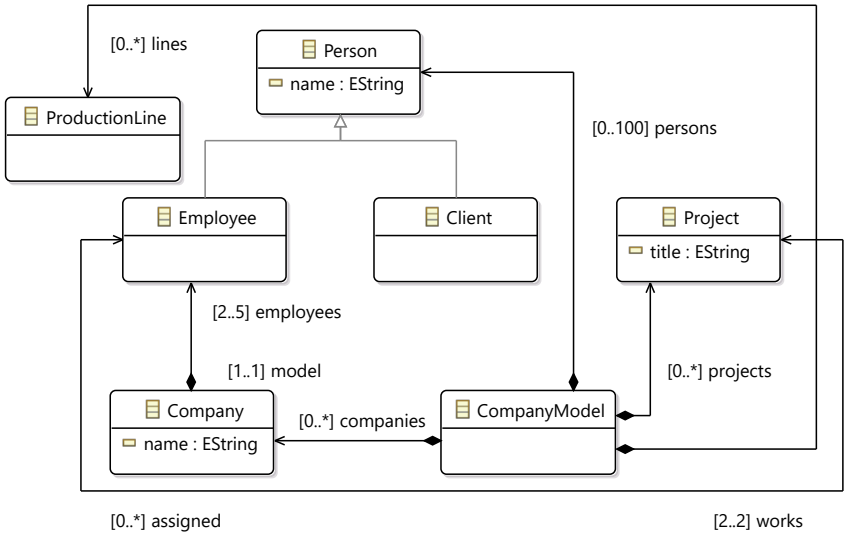

Figure 9 – Model from Fig. 1 prioritizing maintainability, reusability and relaxation index

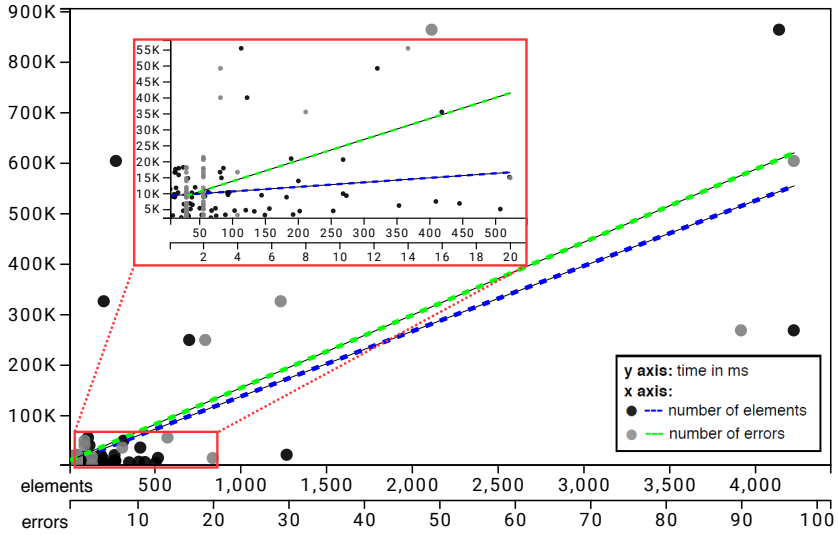

Figure 10 – Relation between repair time in ms per model size and number of errors

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

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

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

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

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

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

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

6 Threats to Validity

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

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

External threats. A potential external threat to the validity of our evaluation is the dataset used for the experiments. We have selected corrupted models resulting in a dataset of 107 models, which may be considered small, however, this threat may be mitigated with the heterogeneity of the sources; these models have been retrieved from different Github repositories and hence from different modelers.

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

7 Related work

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

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

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

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

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

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

8 Conclusions and future work

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[GP01] Marcela Genero and Mario Piattini. Empirical validation of measures for class diagram structural complexity through controlled experiments. In *5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2001.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

About the authors

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

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

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

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