JOT
JOURNAL OF
OBJECT TECHNOLOGY

Addressing the trade off between smells and quality when refactoring class diagrams

Angela Barriga[*], **Lorenzo Bettini**[†], **Ludovico Iovino**[‡], **Adrian Rutle**[*], **and Rogardt Heldal**[*]
[*]Western Norway University of Applied Sciences, Norway
[†]Università degli Studi di Firenze, Italy
[‡]Gran Sasso Science Institute, Italy

ABSTRACT Models are core artifacts of modern software engineering processes, and they are subject to evolution throughout their life cycle due to maintenance and to comply with new requirements as any other software artifact. Smells in modeling are indicators that something may be wrong within the model design. Removing the smells using refactoring usually has a positive effect on the general quality of the model. However, it could have a negative impact in some cases since it could destroy the quality wanted by stakeholders. PARMOREL is a framework that, using reinforcement learning, can automatically refactor models to comply with user preferences. The work presented in this paper extends PARMOREL to support smells detection and selective refactoring based on quality characteristics to assure only the refactoring with a positive impact is applied. We evaluated the approach on a large available public dataset to show that PARMOREL can decide which smells should be refactored to maintain and, even improve, the quality characteristics selected by the user.

KEYWORDS Smells, Refactoring, Quality evaluation, Reinforcement learning.

1. Introduction

Models are becoming core artifacts of modern software engineering processes (Whittle et al. 2014). Models, as happens with code, change and evolve throughout their life cycle due to maintenance and to comply with new requirements. Preserving the quality of these models is of the utmost importance to ease their maintenance and to correctly produce the systems they represent. To this extent, the model-driven engineering (MDE) community has developed a series of mechanisms to identify bad practices and smells that worsen models maintenance and to measure the quality of models.

Smells in code (Beck & Fowler 2018) are not bugs or errors but instead, can be considered as violations of the fundamentals of developing software that decrease the quality of code. In the same way, smells in modeling (Bettini et al. 2019) are indicators that something may be wrong within the model design, even if the model is valid. Some examples of domain modeling smells would be unnecessary duplicated features or classes isolated from the rest of the model, often resulting in uninstantiatable classes, especially if the model is instantiated with a class that could not reach the isolated one. Smells may severely affect the maintenance and evolution of models, as happens with code. Therefore, their early identification and removal is crucial to assure the final quality of models. There are many smells defined in the literature (Mumtaz et al. 2019; Beck & Fowler 2018; Strittmatter et al. 2016) and detecting and removing them is far from trivial. Refactoring models to remove smells might come with a cost. Since removing the smells imply modifying the model structure, this usually has a positive effect on the general quality of the model (Bettini et al. 2019) but, in some cases, it could also have a negative impact. This impact is strictly related to multiple aspects: the model's structural composition, smell occurrences and combinations, etc.

However, to know the impact of the refactoring one needs a way to measure the quality of the model after the refactoring. In this paper, we will use quality characteristics. Quality characteristics have been extensively studied in the literature (Boehm

JOT reference format:
Angela Barriga, Lorenzo Bettini, Ludovico Iovino, Adrian Rutle, and Rogardt Heldal. *Addressing the trade off between smells and quality when refactoring class diagrams.* Journal of Object Technology. Vol. 20, No. 3, 2021.
http://dx.doi.org/10.5381/jot.2021.20.3.a1

et al. 1976; Dromey 1995; Ortega et al. 2003). With them, modelers can quantify how good models are in terms of concepts like analyzability, adaptability, understandability, etc. Several tools exist in code analysis and also in MDE where modelers can define their own characteristics and automatically detect them in models using various automated mechanisms (Basciani et al. 2019; López-Fernández et al. 2014).

The positive effect of calculating these quality characteristics automatically is that they can be used to measure the impact of removing a specific smell on the overall model or on specific quality characteristics (Di Rocco et al. 2014; García-Magariño et al. 2008). By combining the removal of smells and quality measurement, modelers could tackle the refactoring of models to remove smells without compromising the overall model quality, making it possible to find a balance between which smells should be removed and which ones not.

In our previous work (Barriga, Heldal, et al. 2020), we presented PARMOREL, an extensible model repair framework, implemented as an Eclipse plugin, which enables users to deal with different types of model issues and to add their own repair preferences to customize the results. This customization of results is achieved with reinforcement learning (RL) (Thrun & Littman 2000). By using RL, PARMOREL finds the best solution for repairing a model according to the user preferences. So far, as model issues, we tackled with PARMOREL the repair of syntactic errors in broken models. As user preferences, we have worked with quality characteristics (Iovino et al. 2020).

In this paper, we will demonstrate the flexibility of PARMOREL showing that it can support smells detection and refactoring. To achieve this, we integrate PARMOREL with a tool that allows modelers to identify smells and refactor them with known refactorings (Bettini et al. 2019). This extension is based on Edelta (Bettini et al. 2020), a DSL-based tool to define smells and corresponding refactorings in personalized libraries.

To validate this new extension, we solve the trade-off problem between smells and model quality in a dataset used in the literature, consisting of 404 class diagrams extracted from GitHub (Babur 2019). The results are encouraging and show that PARMOREL is able to decide which are the best smells to refactor in order to maintain and, even improve, the quality characteristics selected by the user.

Structure of the paper. This paper is organised as follows: Section 2 illustrates and presents the PARMOREL architecture. Section 3 demonstrates why we need to selectively remove smells instead of addressing all of them. Then, in Section 4, we show the customization applied to PARMOREL to perform selective removal of smells and how existing components have been extended, i.e., with Edelta and with a quality evaluation framework. In Section 5, we evaluate if PARMOREL can success in refactoring with a balance between smells and quality. Then, we present threats to validity in Section 6, explore the related work in Section 7 and conclude the paper in Section 8.

2. PARMOREL Framework

In this section, we briefly present the PARMOREL framework in order to understand its extension (in Section 4) to support selective refactoring of models containing smells. PARMOREL makes use of three main concepts: issues to be found in the models, actions to be applied in response to issues and preferences with which the user customizes how issues are solved. Then, a RL algorithm is in charge of deciding which is the best action to apply in response to an issue, according to preferences selected by the user. The architecture of PARMOREL is based on three main modules that we represented in Fig. 1: a *modeling module*, a *learning module* and a *preferences module*. In the rest of this section, we will explain these components and show how they make the framework flexible to be adapted to the needs of the users in terms of different models, issues, actions, preferences and learning algorithms.

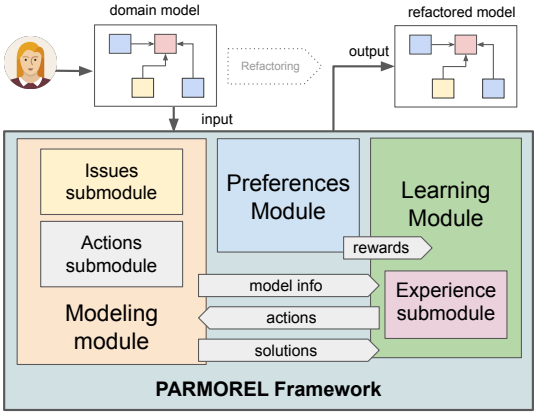

Figure 1 Overview of the PARMOREL architecture

2.1. Modeling module

The *modeling module* is divided in two submodules, namely the *issues submodule* and *actions submodule*.

The *issues submodule* is in charge of identifying which issues are present in the model and sends them to the *learning module*. In (Barriga, Heldal, et al. 2020) we introduced the concept of *issue*. An issue represents something that is improvable in a model regardless of its nature. An issue could be a syntactic or semantic error, a smell, a violation with respect to an architectural pattern or a specific constraint, etc.

The *actions submodule* is in charge of sending to the *learning module* the actions which are available for refactoring the model and of applying the chosen refactorings.

In this paper, we focus on extending the *modeling module* so that PARMOREL supports smell identification and refactoring. Therefore, more details about this module can be found in Section 4.

2.2. Learning module

The *learning module* makes use of RL to learn which actions are the best to refactor the issues in the models according to the preferences introduced by the users.

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. RL allows PARMOREL to perform model

manipulation without having any prior data (i.e., labelled data, historical data, etc.) about removing issues in models.

By using and tuning RL rewards, these algorithms can learn which are the best actions to apply to the model. RL rewards can be adapted to align with any preference introduced by the user as long as it can be quantified, e.g., improving quality characteristics (Iovino et al. 2020). Preferences need to be quantified so that their values can be mapped into RL rewards. For example, the value of the maintainability quality characteristic itself could be used as a reward, if the modeler wants to improve it.

Before finding a refactoring for a given model, PARMOREL is executed for a number of episodes. Each episode equals to one iteration refactoring the model. During the episodes, different actions will be applied to remove the different issues present in the model. For each of these episodes, a refactoring sequence is found, and by applying it, a provisional refactored model is created. The provisional refactored models are analyzed according to the preferences selected by the user, and the result is translated into rewards (e.g., the value of the considered quality characteristic of the refactored model). Hence, PARMOREL can identify how good the applied refactoring is according to the user requirements. Following this process, after each episode, actions leading to the results closest to the user requirements will have higher rewards and thus higher probabilities of being selected. After performing enough refactoring iterations, PARMOREL will select the refactoring with higher rewards and save the final refactored model.

RL is a broad field with many algorithms. In previous work, we compared the performance of different RL algorithms in PARMOREL and Q(λ) was the one that provided us the best performance (Barriga, Mandow, et al. 2020). Hence, we use Q(λ) in our current implementation.

Q(λ) In this algorithm, knowledge acquired is stored in a table structure called Q-Table (Thrun & Littman 2000). This table stores pairs of states (states equal smells in our application) and actions together with a Q-value. The Q-value is calculated using the rewards and it indicates how good each pair is. The Q-value is obtained with repeated calculations based on the Bellman Equation (Bellman 2013) as follows:

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

telling that the maximum future reward is the reward r the agent received for entering the current state s with some action a plus the maximum future reward for the next state s_{t+1} and action a' reduced by a discount factor γ.

This allows inferring the value of the current (s,a) pair based on the estimation of the next one (s_{t+1}, a'), which can be used to calculate an optimal policy to select actions. The factor α provides the learning rate, which determines how much new experience affects the Q-values. One of the variables used to calculate the Q-value, is the maximum weight stored in the Q-table for the next error to refactor ($\max_{a'} Q(s_{t+1}, a')$). This allows us to measure the consequences of applying a certain action in the model (e.g., if applying an action creates a new smell this action would be punished, getting a lower weight). At the end of the execution, pairs with the highest Q-value will conform to the policy to solve the problem. Our algorithm is epsilon-greedy (ε-greedy): it avoids local optima using an exploration-exploitation trade-off by exploring (i.e. choosing a random action) with probability ε, and exploiting (i.e. choosing the action with highest Q-value) the remainder of the time. According to our testing (Barriga, Mandow, et al. 2020), we obtain better results with an ε of 0.3. Regarding other parameters, discount factor (γ), and learning rate (α), we use 1.0 for both of them.

Q(λ) uses a technique called *eligibility traces* (see lines 9-18 in Algorithm 1) to back-propagate the values and received rewards, but it does so not only to the immediately preceding state *e(s,a)* (or pair of state-action), but to all preceding states of the current episode, (stored in the *sae* list, see lines 16-18). The idea is that this propagation decays in intensity the further a state is in the past. This decayed propagation can lead to a speed up in the algorithm's convergence, especially in sparse reward models (Thrun & Littman 2000), which provides rewards only at the end of each episode (e.g., PARMOREL receives the quality characteristics rewards from the provisional refactored model at the end of an episode). The propagation decay is controlled with a parameter λ (see line 18). In practice, the speed of convergence as a function of the value of λ (between 0 and 1) generally has a U-shape. Therefore, the optimal convergence is usually achieved with an intermediate value of λ, which needs to be determined experimentally. According to our experiments (Barriga, Mandow, et al. 2020), we get the best results by giving λ a value of 0.7. Lower or higher values lead to results of lower quality. The new Q-value is temporarily stored in the variable δ (see line 15). It is later stored in the Q-table (see line 17) by adding the already stored Q-value for that pair of state-action (s,a) to the product of α, δ (the new Q-value) and the eligibility trace of (s,a).

The pseudocode depicted in Algorithm 1 is adapted from the one presented in chapter 12 in (Thrun & Littman 2000).

Algorithm 1 Q(λ)

```
1:  Initialize Q-Table
2:  for each episode do
3:      Initialize eligibility table e (default value 0)
4:      Initialize sae as an empty list of state-action pairs
5:      s ← initial state s0
6:      while errors in model != ∅ do
7:          Get state s
8:          Select best action a with ε-greedy policy for s
9:          if a is selected randomly then
10:             reset eligibility to 0
11:             reset sae as an empty list
12:         s_{t+1} ← a applied in s
13:         Add (s,a) to sae list
14:         e(s,a) ← e(s,a) + 1
15:         δ = r + γ max_{a'} Q(s_{t+1}, a') − Q(s,a)
16:         for each s,a in sae do
17:             Q(s,a) = Q(s,a) + αδe(s,a)
18:             e(s,a) ← γλe(s,a)
19:         t ← t + 1
20:         s ← s_{t+1}
```

Experience submodule One of the advantages of using RL is that these algorithms can improve their performance the more they are applied. In our approach, the more PARMOREL modi-

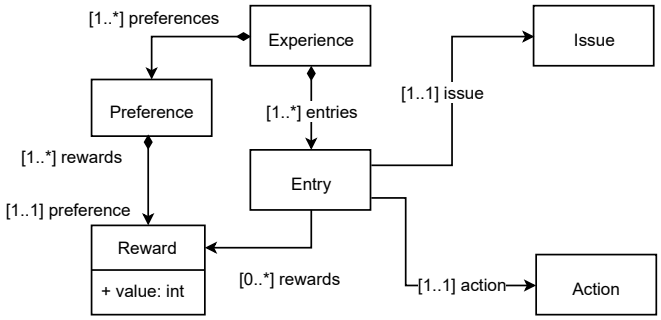

Figure 2 Model of experience in PARMOREL

fies models, the better performance it might get. This is because PARMOREL acquires and builds experience that is reused in later refactorings. To this end, we define the *experience submodule*. This submodule makes use of the machine learning (ML) technique of transfer learning (TL) (Barriga, Rutle, & Heldal 2020). In traditional RL, the value of each pair of issues and actions depends on a single reward; e.g., for a robot learning how to escape a maze, it receives a negative reward when stepping into a wall and a positive one when entering a free space. However, in our case one pair's weight may depend on multiple rewards since it might involve several user preferences, e.g., a user might want to boost the maintainability and reusability of a model. Introducing user preferences complicates reusing the experience acquired by the RL algorithm, since what is a good refactoring for one user might not be acceptable for another one. With this technique, what is learnt from the refactoring of one model could be reused for other models. Hence, consequent executions of PARMOREL could achieve better performance the more experience is reused. Even if the users are different, if the preferences they selected and the issues present in the models are similar, sharing experience would be useful.

We use the model in Fig. 2 to illustrate how PARMOREL supports TL. The learning information gained after each refactoring is represented by the concept Experience which is composed of one to many entries and preferences. The concept Entry refers to the pairs in the Q-table and hence it has references to all the elements that are part of the Q-table: an Issue and an Action. In addition, an Entry has a zero to many references to Reward. The Reward contains a numerical value based on the users' preferences.

The rewards stored in the Experience are used to initialize the Q-table in following executions. This way, if the current user shares any preference with previous ones, the rewards these previous preferences provided in previous refactorings can be used to initialize the new user's Q-table, so that the refactoring does not start from zero. This way, the learning will converge faster and less episodes will be required. When sharing experience in PARMOREL, we reduce the value of ε (see line 8 in Algorithm 1) from 0.3 to 0.15 to enhance the influence of the previous Experience. We initialize the Q-table with the accumulated rewards of the shared preferences multiplied by a discount factor of 0.2. This way we assure previous refactoring processes influence the new ones by jump-starting the process but without interfering with learning new refactoring sequences.

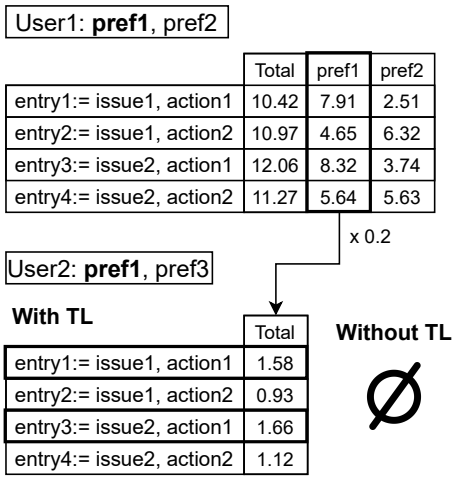

Figure 3 TL between 2 users with a shared preference

Based on our experimental results (Barriga, Rutle, & Heldal 2020), we found that a value of 0.2 gave the best results for our cases. This parameter's value can be modified to affect the impact of previous experience on new refactorings. However, the value should remain a constant during the execution otherwise some parts of the experience will be more favoured than others.

An example of this process is displayed in Fig. 3. In the left part of the image we show the Q-table of *User1* once she finishes using PARMOREL. *User1* chooses as preferences *pref1* and *pref2* to refactor a model with two issues, namely *issue1* and *issue2*. Both issues can be refactored with actions *action1* and *action2*. Then, in the right part of Fig. 3 we show how the Q-table will look for *User2* once she starts using PARMOREL. This user chooses to refactor with preferences *pref1* and *pref3*. The model to refactor is different than the one refactored by *User1*, but since what is relevant for PARMOREL are issues and actions, the *Experience* can be reused regardless of the specific model to refactor. Without TL the Q-table will not exist and a new one will be created, adding more time to the processing part of the learning algorithm. With TL, every entry existent in the *Experience* is copied in the Q-table, and since *pref1* is shared with *User1*, the Q-table is initialized with the rewards provided from this preference multiplied by the discount factor. This way, when PARMOREL starts the refactoring process for *User2*, the time spent in populating the Q-table is reduced and the learning algorithm will already have an intuition of which actions are better for each issue.

For more details about how PARMOREL uses TL and how the *experience submodule* works we refer the reader to our previous work (Barriga, Rutle, & Heldal 2020; Barriga, Heldal, et al. 2020).

2.3. Preferences module

Users can customize the results PARMOREL produces with their own preferences. PARMOREL supports preferences as long as they can be translated into numeric values. PARMOREL will take these values as rewards that will guide the refactor process.

For example, users could prefer to refactor improving a quality characteristic (e.g., maintainability, reusability, understand-

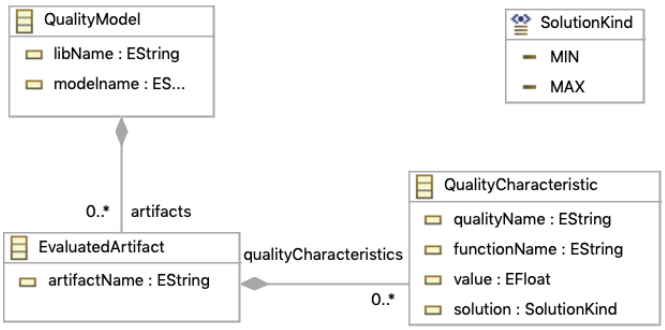

Figure 4 Quality characteristics model

ability, etc.), to minimize the model distance with respect to the original model, etc. PARMOREL will use the rewards to estimate how good or bad each action is to satisfy the user preferences. As part of the preferences given to the users, PARMOREL integrates a quality evaluation tool (Iovino et al. 2020), which is inspired by (Basciani et al. 2016).

Quality Characteristics as preferences This quality evaluation tool supports the specification of quality characteristics conforming to the domain model in Fig. 4. Each EvaluatedArtifact (the artifact from which the quality characteristics will be measured, e.g.; a domain model) will be assigned a set of QualityCharacteristics which can be specified by the modeler. Moreover, whether quality characteristics should be maximized or minimized, is specified in the attribute solution. The calculation function functionName of each quality characteristic has to match with a definition of an EOL (Kolovos et al. 2006) script aggregating the available metrics (as shown in the various formulas) in a predefined library (Basciani et al. 2019). EOL (Kolovos et al. 2006) is 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 it easier to define evaluation operations compared to Java implementations using the EMF API directly (Basciani et al. 2016).

In this paper, we specify the following quality characteristics to be used as user preferences: maintainability, understandability, complexity, and reusability.

The *maintainability* has been defined according to the definition given in (Genero & Piattini 2001) and the formula presented in (Basciani et al. 2016), that is based on some of the metrics shown in Table 1 as follows:

$$Maintainability = \left(\frac{NC + NA + NR + DIT_{Max} + Fanout_{Max}}{5} \right) \quad (2)$$

The definitions of the *understandability* and *complexity* quality characteristics are adopted from (Sheldon & Chung 2006). In particular, *understandability* can be defined as follows:

$$Understandability = \left(\frac{\sum_{k=1}^{NC} PRED + 1}{NC} \right) \quad (3)$$

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.

Characteristic	Acronym
Number of classes	NC
Number of references	NR
Number of opposite references	NOPR
Number of containment references	NCR
Number of attributes	NA
Number of unidirectional references	NUR
Max. generalization hierarchical level	DITmax
Max. reference sibling	FANOUTmax
Number of features	NTF
Sum of inherited structural features	INHF
Attribute inheritance factor	AIF
Number of predecessor in hierarchy	PRED

Table 1 Metrics used in the quality characteristics equations

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:

$$Complexity = (NR - NUR + NOPR + UND + (NR - NCR)) \quad (4)$$

where NUR is the number of unidirectional references measured as the difference between bidirectional and number of references, and UND is the *understandability* value measured as defined in Equation 3.

The *reusability* of a given model can be measured in different ways. One of these is to use the attribute inheritance factor *AIF* as proposed in (Arendt & Taentzer 2013). As presented in (Al-Jáafer & Sabri 2007), AIF can be defined as follows:

$$Reusability = AIF = \left(\frac{INHF}{NTF} \right) \quad (5)$$

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

3. Dilemma: Removing all the smells?

According to (Arendt & Taentzer 2013) a model quality assurance framework should implement three important iterative phases: i) model analysis, ii) identification of smells and iii) removing of the smells. In order to confirm that removing the smells had a positive effect not only formally but also practically, quality evaluation is crucial and can be considered as a litmus test of the refactoring activity. For this reason, it is helpful to evaluate the model before and after removing the smells to see if the applied refactorings effectively improved the design of the model.

In this section, we use an explanatory example to motivate our research. We demonstrate that removing all the smells in a model may be beneficial in terms of quality, but we can have cases in which not all the quality characteristics improve. These

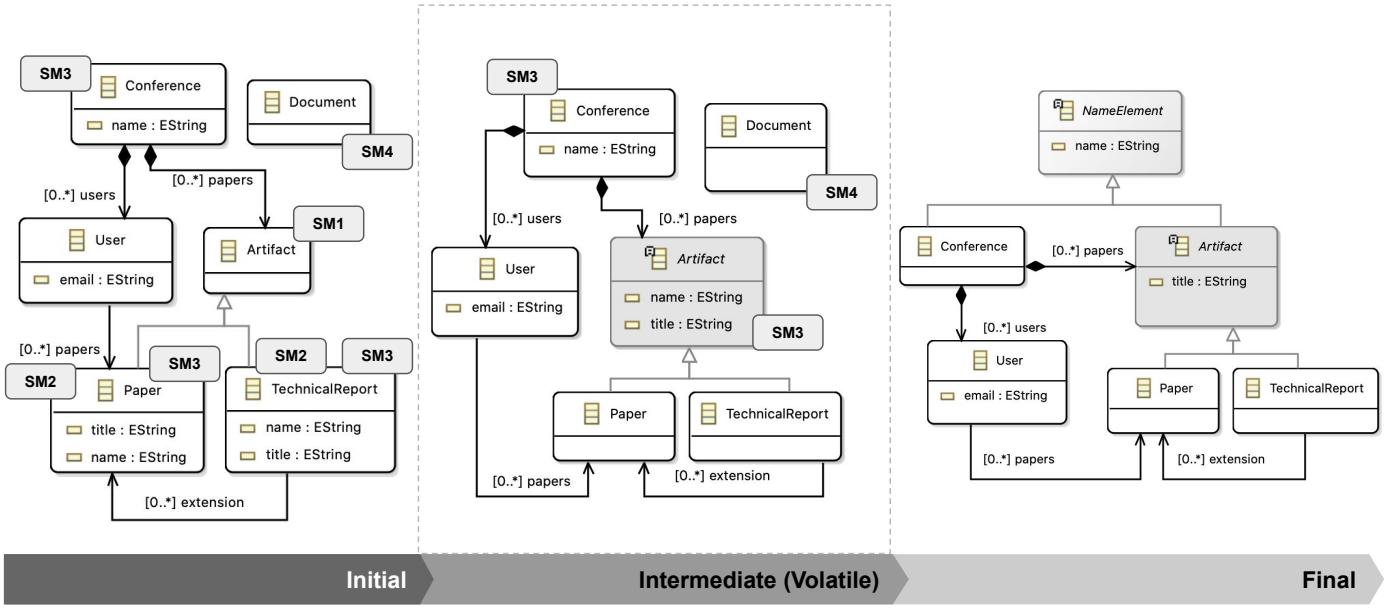

Figure 5 Running example showing an initial smelly model, an intermediate version of the model where SM1 and SM2 are removed, and a final version where SM3 and SM4 are removed

cases strictly depend on the model containing the smells, number of occurrences and the structure of the parts that are not affected by the smells. For instance, evaluating the quality of a huge model with only one smell can give very different results with respect to a small model containing the same smell. Moreover, certain type of smells may affect specific quality characteristics because of the parts of the model they affect (Strittmatter et al. 2016; Basciani et al. 2016; Bettini et al. 2019), as we will see later in this section.

Removing all the smells unconditionally should improve the quality characteristics, but depending on the model structure, smell occurrences and applied refactorings, some of the quality characteristics may get worse. In Fig. 5 we introduce, as an example, a "smelly" model. This domain model is inspired by an example taken from the ATL Zoo and it represents a simplified conference management system that can be used internally by universities or departments. We use this trivial example to highlight the issues and motivate the problem, whereas in Section 5 we will show case studies part of a real dataset.

In this system, as can be seen from Fig. 5 (initial), the modeler can declare a Conference, that contains the submitted Artifacts, that can be identified with a title and a name. Moreover, a set of Users can be defined and registered to the system with their email addresses. In particular, users can be assigned to papers. A Conference contains a set of Artifacts which might be either a Paper or a TechnicalReport, that may extend one or more papers. These two classes extend the class Artifact which is declared as concrete. The possibility of an accidental instantiation of the class Artifact leads to the smell *concrete abstract class* (SM1). Three of the classes in this model, Conference, Paper and TechnicalReport, share the attribute name, which can be identified with the smell SM3, *duplicated features*. The two subclasses of Artifact (i.e., Paper and TechnicalReport) share the attributes title and name (String). This identifies the smell *duplicated features in hierarchy*, i.e., SM2, which is a more specific version of *duplicated features*: here the same feature is found in all of the subclasses of a given superclass. Finally the class Document is declared, maybe with a missing relationship to any of the other classes. This implies the *dead class* smell, i.e., SM4. It is worth noting that the class Document could not be instantiated in a framework like EMF—due to EMF's requirement that all model elements should be contained in a root model element—since Conference is the root of our model. One way to instantiate it is to declare it as root of the model but then the modeler would not be able to instantiate the remaining classes of the metamodel, hence the class is identified as dead.

These four smells may affect multiple quality characteristics, and when multiple smells are automatically removed with refactorings, the quality characteristics can improve but in some cases can also get worse. In the case reported in Fig. 5, four possible refactorings may be applied:

SM1 Concrete abstract class → Make the class abstract
SM2 Duplicated features in hierarchy → Pull up features
SM3 Duplicated features → Extract superclass
SM4 Dead class → Remove class

When our smell finder detects a smell, the corresponding refactoring is immediately applied. Applying all the refactorings immediately after finding the smells might lead to models with lower quality characteristics than the original smelly models. In Fig. 5, the intermediate model shows the model after applying the refactoring for SM1 (Artifact is made abstract) and SM2 (name and title are pulled up into Artifact), whereas the final model in Fig. 5 shows the model after applying all the refactorings, including also a newly created instance of SM3. Hence a new class NameElement is added as superclass for Conference and Artifact) and SM4 is removed by deleting the

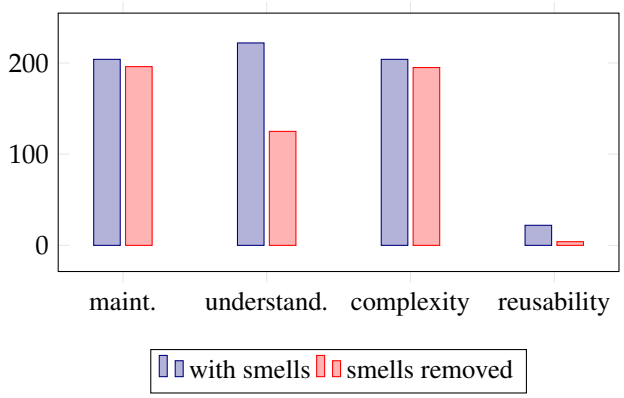

Figure 6 The quality characteristics maintainability (maint.), understandability (understand.), complexity, and reusability before and after removing all the smells in the model in Fig. 5

dead class Document. Although we identified both SM2 and SM3 on the classes Paper and TechnicalReport, we removed the more specific smell SM2. This strategy incorporated in our smells finder and resolver (see Section 4.1) makes sense with respect to how a modeler would refactor this smell, however, it would at the end produce the final model in Fig. 5 which is of lower quality than the original smelly model. Figure 6 displays some quality characteristics, namely, maintainability, understandability, complexity and reusability, measured in the final model before and after the smells are removed. That is, by removing all the smells automatically with the listed refactorings, we will get worse values in all these characteristics except for reusability.

In our example, applying the refactoring associated with SM3 extract superclass, worsen the overall quality of the model, since it affects the maintainability, understandability and complexity by adding a new element into the model. However, SM3 improves the reusability, since it creates more inherited features. The refactorings associated with SM2-SM3 improve the reusability and maintainability by removing features from the model. Removing classes to solve SM4 worsens understandability and complexity while it improves maintainability. Regarding SM1, removing it does not affect the quality characteristics considered in this example, however, if unsolved, it could deteriorate the model's quality in the future, since a class that is concrete when it should be abstract could be incorrectly instantiated.

By combining these refactorings we could face situations where removing several smells lead to no improvement in the model quality, for example by removing SM1 and SM4 we get worse understandability and complexity, without improving any quality characteristics. This is an indication that, by selectively applying refactorings when removing smells, quality characteristics could improve with respect to automatically removing every smell in a model.

To overcome these limitations and complement the automatic approaches as (Bettini et al. 2019; Arendt & Taentzer 2013) we propose a new application of PARMOREL to find a balance between smells refactoring and models quality.

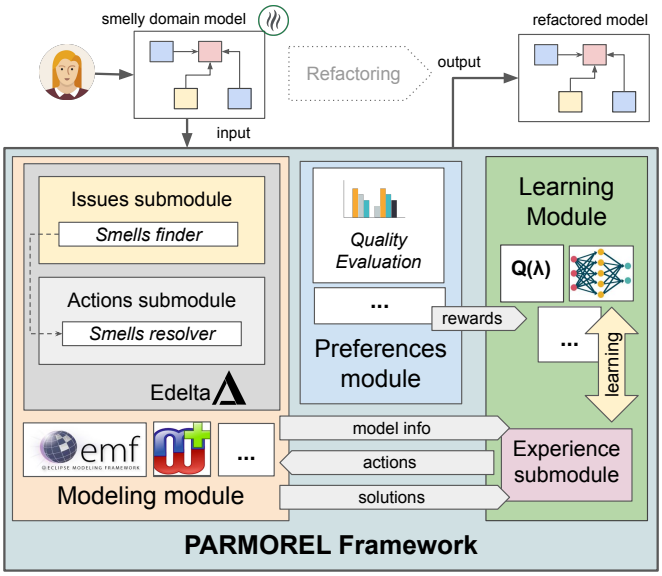

Figure 7 Detailed architecture of the framework

4. Selective smell removal

In previous work (Barriga, Rutle, & Heldal 2020; Barriga, Heldal, et al. 2020; Iovino et al. 2020), we have applied PARMOREL to repair faulty models that violate certain constraints of the Ecore metamodel. In order to apply it for refactoring smells, there are some parts of the framework that must be adapted. In this section, we detail how the PARMOREL framework has been extended to support this task. Figure 7 displays the architecture of the framework in detail after this extension.

4.1. Modeling module extension

In this paper, we extend the modeling module to identify and refactor smells by using EMF (Steinberg et al. 2008) together with Edelta (Bettini et al. 2017) for refactoring Ecore models.

Edelta Edelta is a model refactoring tool, based on a DSL, for easily defining Ecore model evolutions and refactorings. The core features of Edelta and its DSL have been detailed in (Bettini et al. 2017). Edelta provides modelers with constructs for specifying atomic evolutions and complex refactorings. Atomic evolutions are simple changes applied to models, i.e., additions, deletions and edits. Complex refactorings are reusable changes, defined by composing already defined atomic or complex refactoriings. The Edelta DSL has been implemented with Xtext (Bettini 2016) and also a complete IDE based on Eclipse is available, offering syntax highlighting, code completion, error reporting, incremental building, as well as debugging. Recently, in (Bettini et al. 2020), the new version of Edelta was presented, supporting a completely live environment, where the modeler can have an immediate feedback in the IDE of the evolved Ecore models. Edelta has also been used for detecting Ecore model smells and for removing them by means of reusable refactorings organised in libraries (Bettini et al. 2019). In previous work, Edelta has been used as a standalone tool that can be used on a subject metamodel to analyze it, evolve it or to apply refactorings, interactively, with the live IDE environment of Edelta.

Moreover, all these mechanisms can also be used in a standard Java program to process a set of metamodels in batch mode.

In this paper, we make use of Edelta libraries to instantiate the *issues submodule* and the *actions submodule* with a *smells finder* and *smells resolver*, respectively.

Smells finder The *smells finder* uses the Edelta DSL to specify queries for identifying smells in Ecore models. Using the Edelta language, the modeler can provide the specification of custom smell finders and refactorings, which can be properly organized in reusable libraries. The Edelta DSL has a Java-like syntax, so it should be easily understood by Java programmers, but with less "syntactic noise". For example, most of types declarations can be omitted if they can be inferred from the context. Moreover, the Edelta DSL is based on the Java type system and it is completely interoperable with all existing Java types and libraries. Indeed, the types used in the next listings are Java types.

Edelta comes with a smell finder including the smells mentioned in this paper, but the modeler can further extend this library with new smells or refine the existing ones. In Listing 1 we report an extract of an Edelta library containing a few smells finders mentioned in Section 3.

```
def findDuplicatedFeatures(EPackage epackage) {
 return findDuplicatedFeaturesInCollection(
  ePackage.allEStructuralFeatures,
  [existing, current| new EdeltaFeatureEqualityHelper()
   .equals(existing, current) ] )
}
def findDuplicatedFeaturesInCollection(
    Collection<EStructuralFeature> features,
    BiPredicate<EStructuralFeature, EStructuralFeature>
     matcher) {
 val map = newLinkedHashMap
 for (f : features) {
  val existing = map.entrySet.findFirst[matcher.test(it.key
    , f)]
  if (existing != null) {
   existing.value += f
  } else {
   map.put(f, newArrayList(f))
  }
 }
 return map.filter[key, values| values.size > 1]
}
def findConcreteAbstractMetaclasses(EPackage ePackage) {
 return ePackage.allEClasses
  .filter[cl | !cl.abstract && cl.hasSubclasses]
}
...
```

Listing 1 Edelta snippet of the smell finder library

Concerning finding duplicated features, the core function is `findDuplicatedFeaturesInCollection`. This operation takes the collection of features to inspect and a lambda expression[1] that is responsible of deciding whether two features should be considered equal in two different classes[2]. Both `findDuplicatedFeaturesInHierarchy` and `findDuplicatedFeatures` call this operation with a different collection of features to inspect and with a lambda expression that relies on our default implementation of equality detection for features, which scans all the properties of two given features. Note that modelers can reuse `findDuplicatedFeaturesInCollection` with a custom equality matcher for their own new smells and refactorings. The smell finder returns the possible detected duplicated features in an appropriate data structure (in this case, a map). Such a data structure contains the information needed to possibly "resolve" the smell, as shown in the next paragraphs. The definition of `findConcreteAbstractMetaclasses` should be straightforward. In the above code we rely on some utility functions defined in Edelta (e.g., `allEClasses`, `directSubclasses`, etc.) that we do not detail here.

The *smells finder* implements the *issues submodule* in the PARMOREL framework and hence, it takes care of identifying which smells are present in the models and communicating them to the *learning module*.

Smells resolver To complement the *smells finder* we use a *smells resolver* to remove the smells found in the models. When a smell is declared the modeler needs to specify the refactoring to resolve the smell. This correspondence is declared in a Edelta library called *resolver* that basically links the smells with the refactorings. Model refactorings are specified by using the Edelta DSL as well. For instance we could specify that the *duplicated features in hierarchy* should be resolved by *pull up attributes*, that the more general *duplicated features* smell should be resolved by *extract superclass*, and that the *concrete abstract class* should be resolved by simply making the class abstract.

An important feature of Edelta is that it resolves all occurrences of a smell type in one run. For instance, when resolving SM2 in Fig. 5, both of the attributes name and title are pulled up to the same class Artifact. The impact of this batch resolution would be more visible if we had two common superclasses for Paper and TechnicalReport, since in this case, atomic resolutions would lead to potentially pulling up name to one of the superclasses and title to the other one. Listing 3 reports a few Edelta refactorings that we have defined in the catalog published at https://www.metamodelrefactoring.org that we used in the above resolver functions in Listing 2. In particular, in this Listing we show the Edelta operations for *pull up* and *extract superclass* (functions like `addNewEClass` and `addEStructuralFeature` are examples of Edelta atomic refactorings).

```
def resolveDuplicatedFeaturesInHierarchy(EPackage pack) {
 finder.findDuplicatedFeaturesInHierarchy(pack)
  .forEach[superClass, duplicates|
   duplicates.forEach[key, values|
    refactorings.pullUpFeatures(superClass, values) ] ]
}
def resolveDuplicatedFeatures(EPackage pack) {
 finder.findDuplicatedFeatures(pack).values
  .forEach[refactorings.extractSuperclass(it)]
}
def resolveAbstractSubclassesOfConcreteSuperclasses(
     EPackage pack) {
 finder.findAbstractSubclassesOfConcreteSuperclasses(pack)
  .forEach[makeConcrete]
}
...
```

Listing 2 Edelta snippet (i) of the resolver library

[1] In Edelta lambda expressions have the shape: `[ param1, param2, ... | body ]`. As in Java, types of parameters can be omitted when they can be inferred. Note that the lambda expression is assignable to the Java functional interface `BiPredicate`.

[2] We do not report all the code since the complete implementation of the smell finders can be found in the source files of the Edelta plugin: https://github.com/LorenzoBettini/edelta.

```
def extractSuperclass(List<? extends EStructuralFeature>
    duplicates) {
 val feature = duplicates.head;
 val name = feature.name.toFirstUpper + "Element";
 val containingEPackage = feature.EContainingClass.EPackage

 containingEPackage.addNewEClass(name) [
  makeAbstract
  duplicates.map[EContainingClass].forEach[c | c.
     addESuperType(it)]
  pullUpFeatures(duplicates)
 ]
}
def pullUpFeatures(EClass dest, List<? extends
    EStructuralFeature> duplicates) {
 duplicates.head.copyTo(dest)
 removeAllElements(duplicates)
}
...
```

Listing 3 Edelta snippet (ii) of the resolver library

The Edelta DSL supports the definition of new smell finders and refactorings that can be coupled together and thus creating new resolvers (as can be seen in Listing 2). These finder-resolver pairs can be organized in a way which dictates the order in which the smells are resolved by Edelta. Although this order is important, the modeler could also specify mutually-exclusive smell finders since the Edelta specification allows for user-defined libraries. For instance, one could define *duplicated features not in hierarchy* as a counterpart for *duplicated features in hierarchy* so that model elements matched by a former smell finder are not related with model elements matched by a latter one. In this way, an implicit order of resolutions could be defined. All such new smell and resolver definitions are automatically available to the entire ecosystem.

The *smells resolver* implements the *actions submodule*, so it notifies the *learning module* about the refactorings available for each smell. Additionally, it applies the chosen refactorings in the model. As mentioned in Section 3, the integration with an external tool like PARMOREL could also reuse the order of invocation of the resolvers. Since PARMOREL removes smells by their types (e.g. all instances of SM2) we rely on the order in which the smells are found and resolved.

4.2. Issues

Previously, we tackled issues individually. PARMOREL would address them one by one regardless of their type or possible duplicities. This made sense since, in a broken model with syntactic errors, the desirable solution is that all errors are removed. Additionally, these errors could have multiple potential solutions that could modify drastically the model structure.

In our current scenario, we contemplate the possibility to leave smells unsolved as long as this is beneficial for the overall quality of the model. Now, for each smell type, the *smell resolver* provides us with one possible solution. Hence, PARMOREL has to learn whether it is worth it or not to apply the refactoring. According to our testing with the smell types and their refactorings which are implemented for this paper (see Section 5), removing a particular smell type will have a very similar impact on the quality characteristics of the models. This is because the quality characteristics we consider are based on the number of different elements in the models. Changing the number of specific elements with addition or removal or setting values, will affect some quality characteristics positively and others negatively.

Because of this, we consider smells in batches, organized by their types. For example, if a model present 3 instances of SM1 (see Section 3), PARMOREL will tackle SM1 as a batch, deciding to refactor or leave it unsolved, instead of tackling the 3 instances individually.

4.3. Episodes

With this batch organization, we reduce the time needed for refactoring, since the maximum number of episodes the RL algorithm will run depends on the found smell types, and not the smell instances.

As explained in Section 2, an episode equals to one iteration refactoring the model. In each episode, a possible refactoring sequence is found, and by applying it, a provisional refactored model is created. At the end of all the episodes, PARMOREL will have learned which are the best actions to solve the issues in the model according to the user preferences. The maximum number of episodes which PARMOREL runs is a parameter within the framework.

According to our testing, PARMOREL needs, for learning the best refactoring for each model, a maximum of 50 episodes for each present smell type. The more smell types a model contains, the longer PARMOREL will require to learn which is the best refactoring for it. For example, for a model with one smell type, PARMOREL will require a maximum of 50 episodes to converge, while for a model with 5 smell types, the maximum will be 250.

To avoid reaching the maximum needlessly, we run the RL algorithm with an early-stopping criteria. The learning will stop once $max_a Q(s_0, a)$ (the maximum Q-value of the initial state) remains unchanged for 25 episodes.

4.4. Rewards

To support the combination of several quality characteristics as a preference, it is not enough to directly use the values of the characteristics as a reward.

According to the characteristics definitions presented in Section 3, maintainability, understandability and complexity are decreasing characteristics. This means that lower values in these characteristics are an indicator of better quality. By contrast, reusability is an increasing characteristic, meaning that the higher its value is, the better reusability the model has. Hence, we could directly use increasing characteristics values, but decreasing ones need to be converted so that their values can be used as a reward.

For example, a user wants to improve the maintainability and reusability characteristics of a model which initial values (v_0) are 10 and 0.15, respectively. For this model, PARMOREL finds two possible refactorings, R1 and R2, each leading to the following quality values (v_r): R1: maintainability of 9.6 and reusability of 0.02 and R2: maintainability of 9.2 and reusability of 0.17. Maintainability improves in both refactorings while reusability gets better in R2 and worse in R1. If we directly added these values we would obtain a reward of 9.62 for the first refactoring and 9.37 for the second one. With this, PARMOREL

would choose R1 although it worsens reusability rather than choosing R2 which improves both characteristics and gives a better result in maintainability.

To avoid this situation, for every decreasing characteristic, we subtract v_r from v_0 and add v_0 back to the result (see Equation 6). With this, we convert the characteristics values so that the higher they are, the better quality they imply.

There could be situations where different quality characteristics have very different ranges. To avoid that one of the characteristics has more influence on the reward than the others, we transform the values v so that they reflect the improvement each characteristic has undergone within a closer range (see the value x in Equation 7). For example, by applying Equation 6 for R2, where v_r values are 9.2 (decreasing) and 0.17 (increasing), and the v_0 values are 10 and 0.15, respectively, we obtain the v values 10.8 and 0.17. Applying Equation 7 to these values, we obtain the x values 108 and 113.3. Finally, by applying Equation 8, (where n is the number of quality characteristics selected by the user), we add all x values and we obtain the *reward*.

By doing this, the example refactorings would get a reward of 117.3 for R1 and 221.3 for R2. Hence, PARMOREL would choose R2. To make them easier to read, values in Fig. 6 were converted so that higher values imply higher quality by using these equations.

$$v = \begin{cases} (v_0 - v_r) + v_0, & \text{if decreasing characteristic} \\ v_r, & \text{if increasing characteristic} \end{cases} \quad (6)$$

$$x = \frac{v * 100}{v_0}, \text{ if } v == 0 \text{ then } x = 0 \quad (7)$$

$$reward = \sum_{i=1}^{n} x_i \quad (8)$$

5. Evaluation

In this section, we present an evaluation of the proposed approach. In particular, we aim at answering the following research question:

> ***RQ:*** *How well can PARMOREL refactor models with a balance between removing smells and at the same time improving their quality?*

Experiment setup and dataset In this paper, we consider the following quality characteristics (Genero & Piattini 2001): maintainability, understandability, complexity, and reusability. These characteristics are offered to PARMOREL users as preferences. In this experiment, as user preferences, we select to improve maintainability and reusability. These preferences are mapped into reward values as explained in Section 4.4. These characteristics are opposite and, usually, when one improves the other worsens. This adds more challenge to the evaluation, since PARMOREL needs to find a balance for satisfying both quality criteria at the same time. For this evaluation, we choose the dataset from (Babur 2019), containing 555 Ecore models extracted from GitHub. We run PARMOREL in Eclipse 2020-06 (the Modeling package) on a laptop with the following specifications: Windows 10 Home, Intel Core i5-6300U @2.4GHz, 64 bits, 16GB RAM.

Smell	Refactoring
1-Concrete abstract class	Make the class abstract
2-Duplicated features in hierarchy	Pull up features
3-Duplicated features in classes	Extract superclass
4-Dead class	Remove dead class
5-Redundant container relation	Set correct reference as opposite
6-Abstract subclasses of concrete superclass	Make subclasses concrete
7-Abstract concrete class	Make the class concrete
8-Classification by hierarchy	Transform the hierarchy to enum

Table 2 Smells and refactorings supported in the evaluation

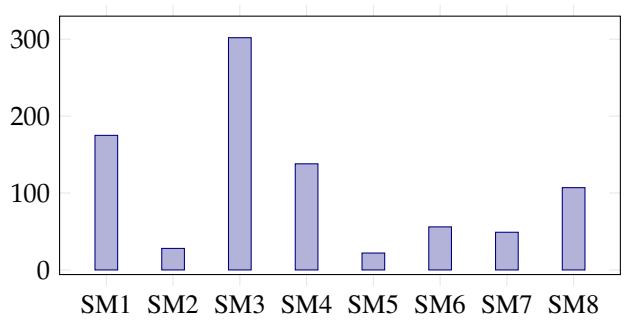

Figure 8 Distribution of smells throughout the dataset

Only 58 models in the dataset do not contain any smell, meaning that 89.54% of the models present some type of smell. From the models with smells, we discard 93, since they are not supported by the quality evaluation tool and hence we can not extract their quality and use it as rewards in the RL algorithm. This makes a total of 404 models subject to be refactored.

The models are of diverse size, containing between 10 and 445 elements, counting classes, attributes and references. From the 8 smell types we have defined in the *smells finder* for this evaluation, each model present between 1 and 7 types. Counting individual instances of each smell type, the models present between 1 and 52 smell instances. Table 2 details the defined smells and the refactoring for each of them. Figure 8 shows the number of models in the dataset containing each smell type.

We randomly split the dataset of models with an 80-20% distribution, refactoring 20% of the models twice, with and without having first refactored the 80%. With this, we analyze the impact of reusing learning with the *experience submodule* on the refactoring time of the 20%.

Analysis of results When refactoring the 80+20% of the dataset, it takes PARMOREL between 0.9 and 56.5s to learn how to refactor each model.

When refactoring the 20% independently, without reusing learning, it takes PARMOREL an average of 37% more time to refactor these models. Faster refactoring happens in models with bigger size, since the bigger the models, the more learning can be reused from previous refactorings. By comparing the refactor time from refactoring with and without reusing learning, we can conclude that PARMOREL streamlines the refactor time of the models between 2% and 61% when it has learned from refactoring other models.

Regarding maintainability, PARMOREL is able to improve it in 33.6% of the models. For 40% of the models it remains unchanged and, for the remaining 26.3%, it worsens. For reusability, 74.50% of the models present better results after refactoring and 25.2% remains unchanged. Only one model from the dataset presented worse reusability after refactoring. These results are summarized in Fig. 9. In 100% of the models, whenever one of the characteristics worsened, the other one improved. These refactorings were selected because in the trade-offs between the characteristics values the best decision for the overall quality of the model was to worsen one of the characteristics in benefit of the other one.

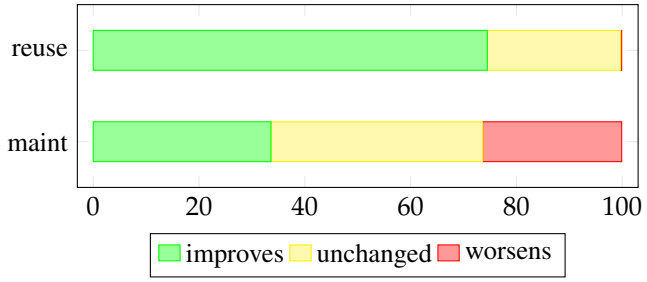

Figure 9 Reusability and maintainability results

Figure 10 displays the percentage of each smell type removed from the total present in the models for which maintainability and reusability improves, respectively. SM1-3 are mostly removed in both cases, while SM4-8 are mostly ignored. Moreover, SM4 and SM8 are more often removed when reusability improves, mostly because the refactoring of these smells reduce the total number of elements in the model.

Taking into account the characteristics in combination, PARMOREL was able to improve the quality of both of them in 31.43% of the models in the dataset. It also improves one of the two characteristics in 45.29% of the models while for 23.28% both remained unchanged (see Fig. 11).

As a conclusion, only in 22.27% of the models the best solution found by PARMOREL was to remove all the smells With the results of this evaluation, we can conclude that when taking into account the quality of the models, the best solution is usually not to remove all the smells. Hence, as an answer to

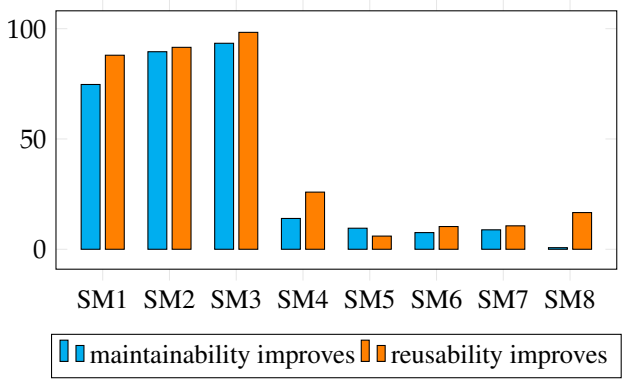

Figure 10 Percentage of each smell type fixed when quality improves

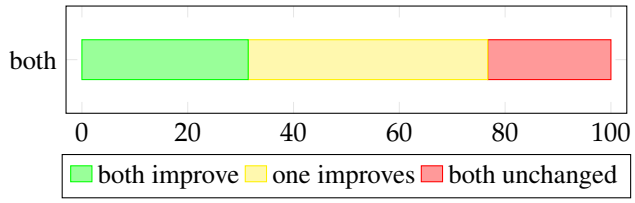

Figure 11 Both characteristics results after refactoring

our research question, PARMOREL is able to refactor the models with a balance between which smells should be addressed without degrading the quality of the models and even improving it. In most cases, the refactored model presents higher quality in the characteristics selected by the user than the original one (76.72% of the models in the evaluation). Additionally, as Fig. 10 shows, PARMOREL has the tendency to remove some of the smell types and to ignore others.

6. Threats to validity

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

Internal validity Internal threats are factors influencing the outcomes of the performed experiment. One potential internal threat is that we focus on automatically detectable smells and this could limit the applicability of the approach since semantic-driven smells might not be representable with the Edelta DSL syntax. Moreover the correctness of the experiments results are driven by the solver, the applied refactoring, smells definitions and quality characteristics calculation formulas. All these elements are defined by modelers and then subject to possible inconsistencies that could influence the final result. To mitigate this aspect we reused, when possible, existing definitions from literature and represented them faithfully with the corresponding models or DSL syntax.

External validity In this context we discuss how the conducted experiment would still be valid outside the used setting. To mitigate this aspect, we considered various models since the dataset is heterogeneous and used in other experiments in literature (Nguyen et al. 2019). We plan to further replicate the experiment with other large datasets.

Throughout the paper we have picked four quality characteristics as a proof of concept to measure the quality of the refactored models. Likewise, we work with a set of eight smells and their corresponding refactorings. Many other characteristics could be measured in the models and other smells could be identified together with different refactorings. We consider the set of characteristics, smells and refactorings representative enough since they are related to different elements in the models, covering a wide range of structural changes in them.

Finally, the examples in the paper are based on EMF and Ecore models, but as we explained, it is possible to switch to other modeling frameworks by extending PARMOREL. Within EMF, the work presented in this paper is specific for Ecore models. However, it could be applied in general to models

instances if the refactoring actions retrieved from the framework were domain specific.

7. Related Work

This section discusses relevant works that are related to smells detection and code refactoring with ML, model refactoring, ML approaches for MDE and, recommender systems.

Smells detection and code refactoring with ML ML for smells detection has been more applied at code level than at model level. In (Fontana et al. 2016), the authors perform a comparative study with different ML techniques for identifying a set of four smells. They achieve high accuracy without needing much data for each smell. However, in the literature review presented in (Azeem et al. 2019) and (Di Nucci et al. 2018), authors point that most studies are done at a theoretical level, and there are still big open challenges the field needs to overcome to reach its full potential.

ML offers the possibility to identify complex smells and it could also be used to detect smells in models. However, users would need to find or define their own datasets in order to tackle the smells they are interested in. The scope of the smells detected using the Edelta DSL is automatically detectable smells, but users just need to define their own smells at code level without needing to train on any dataset. Regarding code refactoring, different ML techniques (Alenezi et al. 2020; Sheneamer 2020) have been applied to predict and identify which parts of the code are prone to be refactored. By doing so, the time spent in refactoring can be reduced. Although our current approach does not support predictions, we use RL to identify both the parts of models that should be refactored in their current state and what is the best action to perform the refactoring. Approaches for code refactoring usually rely on great amounts of data, including code's historic evolution coming from public code repositories. This amount and type of data is not yet available in the MDE field.

Model refactoring The concept of refactoring has been explored using UML class diagrams in (Mens 2006) after a complete analysis of Fowler in (Fowler 1999) for code. A DSL called *Wodel* (Gómez-abajo et al. 2016), allows to create model mutations by means of a metamodel independent specification. Creation, deletion and reference reversal are the primitives offered by the model mutations whereas the composition of mutations are similar to the Edelta mechanism. The specifications are translated into Java code but Edelta works in a different abstraction layer in which the refactoring / mutation is applied.

Similarly to the applied refactorings used in the experiments, a refactoring catalog for UML models is presented in (Sunyé et al. 2001). Whereas in (Xing & Stroulia 2006; Fadhel et al. 2012) mechanisms for detecting refactorings are presented. Lastly, the approach in (Langer et al. 2013) proposes an a searching algorithm for occurrences of composite operations within a set of detected atomic changes in a post-processing manner.

In these approaches, the user is responsible of deciding which refactorings to apply in the model and sometimes of designing them. In PARMOREL, we abstract users from this burden as the tool will take care of deciding which refactorings should be applied to satisfy the user preferences.

ML approaches for MDE We could not find in the literature any research applying RL to model refactoring hence, we focus on other ML techniques as related work.

Puissant et al. propose a tool called Badger based on an artificial intelligence technique called automated planning (Puissant et al. 2015). Badger generates sequences that lead from an initial state to a defined goal.

It has a set of repaired operations to which users can assign costs and weights to decide its priority. Badger generates a set of plans, each plan being a possible way to repair one error. This makes it difficult for the user to decide which action to apply without knowing how it affects the rest of the model. We prefer to generate alternative sequences to refactor the whole model since some actions can modify the model drastically.

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 shown promising results dealing with model transformations and evolution scenarios, for example in (Kessentini et al. 2017) 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, in the beginning, less efficient in large state spaces, it can compensate with its learning capability. In the beginning, performance might be poor, but with time refactoring becomes straightforward.

Some approaches make use of neural network (NN) architectures to solve different MDE problems. In (Burgueño et al. 2019) authors present a NN architecture for model transformation without specifying code for any specific transformations. Tackling model refactoring, in (Sidhu et al. 2020) authors make use of a deep NN architecture to refactor UML diagrams with symptoms of design flaws. NN need a great amount of data in order to work. The produced solutions are tightly related to the training dataset, so if the requirements of the problem changes, so needs to do the data. By using RL we do not need training data, as these algorithms learn by directly interacting with the models and, by using the abstract concepts of PARMOREL architecture, our tool can easily be adapted to solve different problems without the burden of designing new datasets.

Recommender systems Other approaches such as (Cuadrado et al. 2018; Muşlu et al. 2012) work as recommender systems (both for code and models) instead of only relying on automation. PARMOREL may also be utilized like a recommender system allowing users to choose the solution they prefer from a ranked list of proposed solutions. These choices are in turn fed back to the learning algorithm and affect the rewards (Barriga, Heldal, et al. 2020). However, the main focus of this paper is on providing automatic model refactoring to remove smells. Hence, instead of letting the users know about the consequences of the refactorings so that they decide a solution, we ask them beforehand which consequences (quality characteristics as preferences) they prefer, and we use these preferences to guide the refactoring phase. In (Cuadrado et al. 2018) authors present a

catalogue of quick fixes, knowing which one of them can solve each problem. In our paper, each smell found by the smell finder has a corresponding refactoring, however, we have worked in scenarios where we had a set of available actions and we did not know which one solved each smell. Finally, although some quick-fix approaches (Cuadrado et al. 2018) might be initially faster than PARMOREL, the idea of our approach is that it learns and streamlines its performance the more models it refactors. As could be seen in the evaluation, with a relatively small dataset we already were able to refactor models in which the issues were known by PARMOREL 37% faster on average.

8. Conclusions and future work

In this paper, we present a new PARMOREL extension to support smells detection and selective refactoring. The approach is able to selectively remove smells that has impact on the quality characteristics expressed as preference by the user. To achieve this, we integrate PARMOREL with a tool that allows modelers to identify smells and refactor them with precise refactorings. This extension is based on the integration of tools, e.g., Edelta, and a model-based quality assessment methodology. We demonstrated how we can solve the trade-off between smells and quality characteristics with a dataset used in the literature, consisting of 404 models extracted from GitHub. The results are positive and show that PARMOREL effectively select the best smells to refactor in order to maintain and, even improve, the quality characteristics expressed by the modeler. We outline that this approach is totally model-based and that can be further extended with other preferences, issues and actions that we plan to investigate. The main strength of PARMOREL is the degree of flexibility it provides to the user.

In this flexible environment, we use reinforcement learning to learn how to refactor a model without any prior knowledge of the model, and by using our transfer learning approach with experience sharing, we can forward what the framework learns from previous refactorings. Reinforcement learning might have the weakness to provide a slower solution than other approaches during the first refactorings, however, the idea of our approach is that the learning module learns and streamlines the performance the more models it refactors.

Currently, PARMOREL is limited to quantitative user preferences and it needs to get a set of actions to modify the model, unlike other approaches these actions cannot yet be inferred from the issues in the models. Also, PARMOREL needs to detect issues in a model in order to improve it, it cannot deal with models without issues yet. We plan to address these limitations as part of our future work.

Next, we plan to create a benchmark using different model datasets, including the one used in this paper, with which we will compare PARMOREL results and its performance to other existing model refactor and repair approaches in the literature. Also, we plan to extend PARMOREL to solve other problems relevant in the modeling field, like model refactoring after their corresponding metamodel evolves (co-evolution) and making architectural models compliant with best practices and recommended design patterns.

Additionally, we plan to extend the learning module with other algorithms beyond reinforcement learning, specially focusing in other AI and search-based approaches and study their performance with respect to RL algorithms.

References

Alenezi, M., Akour, M., & Al Qasem, O. (2020). Harnessing deep learning algorithms to predict software refactoring. *Telkomnika*, *18*(6).

Al-Jáafer, J., & Sabri, K. E. (2007). *Metrics for object oriented design (mood) to assess java programs* (Tech. Rep.).

Arendt, T., & Taentzer, G. (2013). A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering*, *20*(2), 141–184.

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, *108*, 115–138.

Babur, O. (2019, March). *A labeled Ecore metamodel dataset for domain clustering.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.2585456

Barriga, A., Heldal, R., Iovino, L., Marthinsen, M., & Rutle, A. (2020). An extensible framework for customizable model repair. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 24–34). ACM.

Barriga, A., Mandow, L., Perez de la Cruz, J. L., Rutle, A., Heldal, R., & Iovino, L. (2020). A comparative study of reinforcement learning techniques to repair models. In *2020 ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C).* ACM.

Barriga, A., Rutle, A., & Heldal, R. (2020, July). Improving model repair through experience sharing. *Journal of Object Technology*, *19*(2), 13:1-21.

Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2016). A customizable approach for the automated quality assessment of modelling artifacts. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 88–93). IEEE.

Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2019). A tool-supported approach for assessing the quality of modeling artifacts. *Journal of Computer Languages*, *51*, 173–192.

Beck, K., & Fowler, M. (2018). Bad smells in code. In *Refactoring: Improving the design of existing code* (2nd ed., chap. 3). Addison-Wesley.

Bellman, R. (2013). *Dynamic programming*. Courier Corporation.

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend* (2nd ed.). Packt Publishing Ltd.

Bettini, L., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2017). Edelta: An approach for defining and applying reusable metamodel refactorings. In *Proc. MODELS (Satellite Events)* (pp. 71–80). ACM.

Bettini, L., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2019). Quality-driven detection and resolution of metamodel smells.

IEEE Access, *7*, 16364–16376.

Bettini, L., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2020). Edelta 2.0: Supporting live metamodel evolutions. In *Proc. MODELS (Satellite Events)* (pp. 1–10). ACM.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on software engineering* (pp. 592–605). IEEE Computer Society Press.

Burgueño, L., Cabot, J., & Gérard, S. (2019). An LSTM-Based Neural Network Architecture for Model Transformations. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 294–299). IEEE.

Cuadrado, J. S., Guerra, E., & de Lara, J. (2018). Quick fixing atl transformations with speculative analysis. *Software & Systems Modeling*, *17*(3), 779–813.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018). Detecting code smells using machine learning techniques: are we there yet? In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)* (pp. 612–621). IEEE.

Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2014). Mining metrics for understanding metamodel characteristics. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering* (pp. 55–60). ACM.

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on software engineering*, *21*(2), 146–162.

Fadhel, A. B., Kessentini, M., Langer, P., & Wimmer, M. (2012). Search-based detection of high-level model changes. In *Icsm* (pp. 212–221). IEEE Computer Society.

Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*(3), 1143–1191.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley.

García-Magariño, I., Gómez-Sanz, J. J., & Fuentes-Fernández, R. (2008). An Evaluation Framework for MAS Modeling Languages Based on Metamodel Metrics. In *AOSE* (Vol. 5386, pp. 101–115). Springer.

Genero, M., & Piattini, M. (2001). Empirical validation of measures for class diagram structural complexity through controlled experiments. In *QAOOSE@ECOOP*.

Gómez-abajo, P., Guerra, E., Lara, J. D., Gomeza, P., & Guerra, E. (2016). Wodel : A Domain-Specific Language for Model Mutation. In *SAC* (pp. 1–6). ACM.

Iovino, L., Barriga, A., Rutle, A., & Heldal, R. (2020). Model repair with quality-based reinforcement learning. *Journal of Object Technology*, *19*(2), 17.

Kessentini, M., Mansoor, U., Wimmer, M., Ouni, A., & Deb, K. (2017). Search-based detection of model level changes. *Empirical Software Engineering*, *22*(2), 670–715.

Kolovos, D. S., Paige, R. F., & Polack, F. A. (2006). The epsilon object language (EOL). In *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 128–142). Springer.

Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., & Kappel, G. (2013). A posteriori operation detection in evolving software models. *J. Syst. Softw*, *86*(2), 551–566.

López-Fernández, J. J., Guerra, E., & De Lara, J. (2014). Assessing the Quality of Meta-models. In *MoDeVVa@MODELS* (pp. 3–12). CEUR-WS.org.

Mens, T. (2006). On the use of graph transformations for model refactoring. In *GTTSE (Revised Papers)* (pp. 219–257). Springer.

Mumtaz, H., Alshayeb, M., Mahmood, S., & Niazi, M. (2019). A survey on UML model smells detection techniques for software refactoring. *Journal of Software: Evolution and Process*, *31*(3).

Muşlu, K., Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. (2012). Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices*, *47*(10), 669–682.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Pierantonio, A., & Iovino, L. (2019). Automated Classification of Metamodel Repositories: A Machine Learning Approach. In *MODELS* (pp. 272–282). IEEE.

Ortega, M., Pérez, M., & Rojas, T. (2003). Construction of a systemic quality model for evaluating a software product. *Software Quality Journal*, *11*(3), 219–242.

Puissant, J. P., Van Der Straeten, R., & Mens, T. (2015). Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, *14*(1), 461–481.

Sheldon, F. T., & Chung, H. (2006). Measuring the complexity of class diagrams in reverse engineering. *Journal of Software Maintenance*, *18*(5), 333–350.

Sheneamer, A. M. (2020). An Automatic Advisor for Refactoring Software Clones Based on Machine Learning. *IEEE Access*, *8*, 124978–124988.

Sidhu, B. K., Singh, K., & Sharma, N. (2020). A machine learning approach to software model refactoring. *International Journal of Computers and Applications*, 1–12.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework* (2nd ed.). Addison-Wesley.

Strittmatter, M., Hinkel, G., Langhammer, M., Jung, R., & Heinrich, R. (2016). Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. In *CEUR Workshop Proceedings* (Vol. 1706, p. 30-39). CEUR.

Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.-M. (2001). Refactoring UML Models. In *Proceedings of UML* (Vol. 2185, pp. 134–148). Springer.

Thrun, S., & Littman, M. L. (2000). Reinforcement learning: an introduction. *AI Magazine*, *21*(1), 103–103.

Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE software*, *31*(3), 79–85.

Xing, Z., & Stroulia, E. (2006). Refactoring Detection based on UMLDiff Change-Facts Queries. In *WCRE* (pp. 263–274). IEEE.

About the authors

Angela Barriga is a PhD Candidate at Western Norway University of Applied Sciences. She has experience working with machine learning, computer vision, gerontechnology and pervasive systems. Barriga's thesis is focused on model repair, 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.

Lorenzo Bettini is an Associate Professor in Computer Science at DISIA Dipartimento di Statistica, Informatica, Applicazioni 'Giuseppe Parenti', Università di Firenze, Italy, since February 2016. Previously, he was an Assistant Professor (Researcher) in Computer Science at Dipartimento di Informatica, Università di Torino, Italy. His research interests cover design, theory and implementation of DSLs and programming languages (in particular Object-Oriented languages and Network aware languages). He is also committer of the Eclipse projects Xtext and SWTBot and the project lead of the Eclipse project EMF Parsley. Contact him at lorenzo.bettini@unifi.it, or visit http://www.lorenzobettini.it.

Ludovico Iovino is Assistant Professor at the GSSI Gran Sasso Science Institute, LAquila - 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.

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

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