A Metamodel for the Specification and Verification of Model Refactoring Actions*

Davide Arcelli
University of L'Aquila
L'Aquila, Italy
davide.arcelli@univaq.it

Vittorio Cortellessa
University of L'Aquila
L'Aquila, Italy
vittorio.cortellessa@univaq.it

Daniele Di Pompeo
University of L'Aquila
L'Aquila, Italy
daniele.dipompeo@univaq.it

ABSTRACT

Refactoring has become a valuable activity during the software development lifecycle, because it can be induced by different causes, like new requirements or quality improvement. In code-based development contexts this activity has been widely studied, whereas in model-driven ones, where models are first-class development entities, there are many issues yet to be tackled. In this paper we present a metamodel that supports the specification of pre- and post- conditions of model refactoring actions, and the automated derivation and verification of such conditions in specific modeling languages. Our work is aimed at helping users to implement refactoring actions in the adopted modelling language by providing an environment for guaranteeing the feasibility of refactoring actions. Our primary focus is on the definition of applicable sequences of refactoring actions, rather than on the user-driven step-by-step application of refactoring actions. As an example, we illustrate the applicability of our metamodel for UML models refactoring.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; *Software maintenance tools*;

KEYWORDS

Model-Driven Engineering, Refactoring, Language-independent refactoring metamodel, Refactoring feasibility

ACM Reference Format:
Davide Arcelli, Vittorio Cortellessa, and Daniele Di Pompeo. 2018. A Metamodel for the Specification and Verification of Model Refactoring Actions. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Refactoring is acquiring ever more relevance during software lifecycle due to fast and frequent software changes that require flexibility in each development phase. As a consequence, different code-based refactoring approaches have recently appeared in literature, as based on previously developed concepts.

For exaemple, Opdyke had introduced in his Ph.D. thesis the feasibility of a sequence of refactoring actions by considering and introducing pre- and post-condition aspects [20]. Lately, Fowler et al. had provided a dictionary of refactoring actions aimed at driving users selection [10]. By leveraging on Opdyke's approach, Ò Cinnèide et al. have introduced a framework for automatically verifying the feasibility of a sequence of refactoring actions [17]. In the latter, they have formalised whether a sequence is feasible or not by reducing and validating its pre- and post-condition.

In our recent work [2, 3], we have targeted the need of implementing refactoring actions in model-based approaches. In this context, we have experimented a lack of support especially to the definition of sequences of actions that can be batch-applied, for example in the context of evolutionary algorithms, without user interactions. In order to start targeting this lack, in this paper we present a metamodel, that we have developed by leveraging on Opdyke and Ò Cinnèide concepts, for supporting refactoring activities, with a special focus on pre- and post-condition specification and verification.

We have conceived our metamodel with the aim of being applied to multiple modelling languages. Thus, we have introduced different anchor points for applying the metamodel to their modelling languages. Furthermore, we have developed a set of external helpers aimed to execute common operations, for example, the reduction of the sequence pre-condition or the validation of the post-condition. Moreover, we have decided to use OCL as constraint language because it is quite commonly adopted in MDE ecosystems.

We have applied our metamodel to UML and Æmilia [7]. For both languages, we have implemented different refactoring actions to build feasible sequences of refactoring actions. Due to space limitations, we report here only the UML implementation.

The remaining of this paper is structured as follows: Section 2 introduces the related work; in Section 3 we describe our metamodel, while in Section 4 we illustrate its application to UML; conclusion and future work are reported in Section 5.

2 RELATED WORK

Software refactoring is a well-known activity very likely born with programming. The fix-it-later approach, where fixing problems (especially non-functional ones) is demanded after a prototypal version of the code has been produced, implies refactoring as a first-class activity, interleaved with other phases of the software lifecycle. Although code refactoring is an extensively explored topic in literature [1, 16, 21, 24], the wide adoption of MDE techniques in software design and development, which has led models to be

*This work has been partially funded by the ECSEL project MegaMart2 (grant agreement No 737494).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

first-class artefacts, paved the way to model refactoring, which nowadays represents a challenging research area [4, 10, 12, 23].

Recently, several approaches have tried to improve code refactoring within IDEs (eg, Eclipse, IntelliJ Idea). Kim et al. have proposed a series of papers [13, 14] targeting Java code refactoring through the implementation of some design patterns defined in [11]. Our approach differs in that, first, it supports model-based refactoring in a language-independent way and, second, it leaves to the user the definition of refactoring actions.

In the model-based context, different approaches have been published, which share the characteristic of being user-driven. EMF Refactor [5] provides a suite of well-known refactoring actions by exploiting a statical analysis of the target model and by considering design quality metrics. Operation Recorder [8] allows to derive model refactoring, based on model differences, by applying a pattern matching algorithm on source and refactored models. With respect to these approaches, we natively enable the specification of sequences of refactoring actions.

Our approach borrows some concepts from code refactoring approaches [17, 20], by providing support to specify and verify pre- and postconditions of model refactoring actions and to automatically derive and verify pre- and postconditions for sequences of refactoring actions.

Fowler et al. have introduced in [10] a cornerstone for code refactoring. They have presented a dictionary of refactoring actions, which helps developers in selecting the most valuable actions that solve their issues.

Ouni et al. have presented an approach for fixing code smells by suggesting design patterns [21]. They have used a multi-objective function to drive the search within the solution space. By exploiting the QMOOD (Quality Model for Object-Oriented Design) [6] model, they are able to calculate the effectiveness of each refactoring action on the quality of the source code. Moreover, for each refactoring operation they have used a pre-condition style as Opdyke had introduced in [20]. Basically, our approach takes into account the last aspect, by trying to formalise the Opdyke's concepts in a model-driven refactoring environment.

The main differences between code-based and model-based refactoring come form the different nature of the refactoring targeted artefacts.

The main issue in model-based refactoring is to maintain the conformance among views. For example, a refactoring action might directly change a view (e.g. static view), but it might indirectly involve other views (e.g. the behavioral one). Mansoor et al. have presented an example that intends to address this issue [15], as they have proposed a solution for maintaining intra-view conformance. In this direction, our approach provides a solution in which we can verify the feasibility of the sequence by evaluating the postcondition of a refactoring sequence.

3 METAMODEL FOR MODEL REFACTORING

In this section, we describe the core of our work by: (i) introducing the metamodel for model refactoring (Section 3.1), and (ii) describing the metamodel facilities that we provide for enabling users to "plug" refactoring actions into a language-dependent context, thus inducing pre- and post- conditions support within the target modelling language (Section 3.2).

Several model-based refactoring approaches exist in literature, but none of them is based on a general metamodel that supports refactoring feasibility in multiple modeling languages. In fact, our metamodel has to be intended as an instrument that allows to generate a set of facilities (data-structures and algorithms) that ease the verification of refactoring applicability, as described in the remaining of this section.

3.1 Metamodel Description

Our metamodel has been defined within the Eclipse Modelling Framework [22], hence it basically consists of an (annotated) ecore file accompained by a .genmodel file that allows to generate all the required facilities. Figure 1 depicts the metamodel big picture.

The `Refactoring` metaclass (bottom-left of Fig. 1) represents the root, although we would not generate any model out of this metamodel, as intended in a "pure" MDE context. In fact, key-applications of our framework fall within contexts where a large number of refactoring action sequences have to be automatically generated, applied and verified, as for example in multi-objective optimization contexts [2].

A `Refactoring` may contain:

- A sequence of refactoring `Actions`, namely *actions*;
- A `Precondition` and a `Postcondition`, both representing the general concept of `Condition`, which basically contains a logical formula in first-order logics, namely `FOLSpecification` (see the *conditionFormula* reference).

`Refactoring` `Pre-` and `Post- Conditions` are derived from `Pre-` and `Post- Conditions` of the `Actions` composing the `Refactoring`.

The concepts of pre and post- condition are borrowed from theoretical code-based aspects initially presented by Opdyke [20], and successively improved by Ó Cinnéide et al. [17]. In the latter, the **feasibility** of a sequence of refactoring actions (i.e. a `Refactoring`) is guaranteed if and only if the conditions of each action do not break the applicability of the remaining actions in the sequence. In practical terms, the precondition of the first action has to be verified, then its postcondition does not have to break the precondition of the second action in the sequence, and so on, for each action in the sequence.

For example, let us define the following refactoring actions in a UML context:

(1) *moveComp(c :* `Component`*, T : Set(*`Node`*))*, which re-deploys a `Component`, namely *c*, to a given set of target `Nodes`, namely *T*.
(2) *deleteComp(c :* `Component`*)*, which properly removes a `Component`, namely *c*, from the UML model ([1]).

Their pre- and post- conditions, namely pre_1, $post_1$, pre_2 and $post_2$, would be as follows:

- $pre_1 = (\exists\ c \in C) \wedge (\forall\ t \in T\ \exists\ t \in N)$, where C and N are the set of all the `Components` and `Nodes` in the UML model, respectively. Hence, pre_1 checks the existence of the moved `Component` and each selected target `Nodes`.

[1] By *properly*, here we implicitly assume that *deleteComp* action takes into account intra-model consistency issues, e.g. deleting a `Component` implies that all its deployment relations have to be deleted as well.

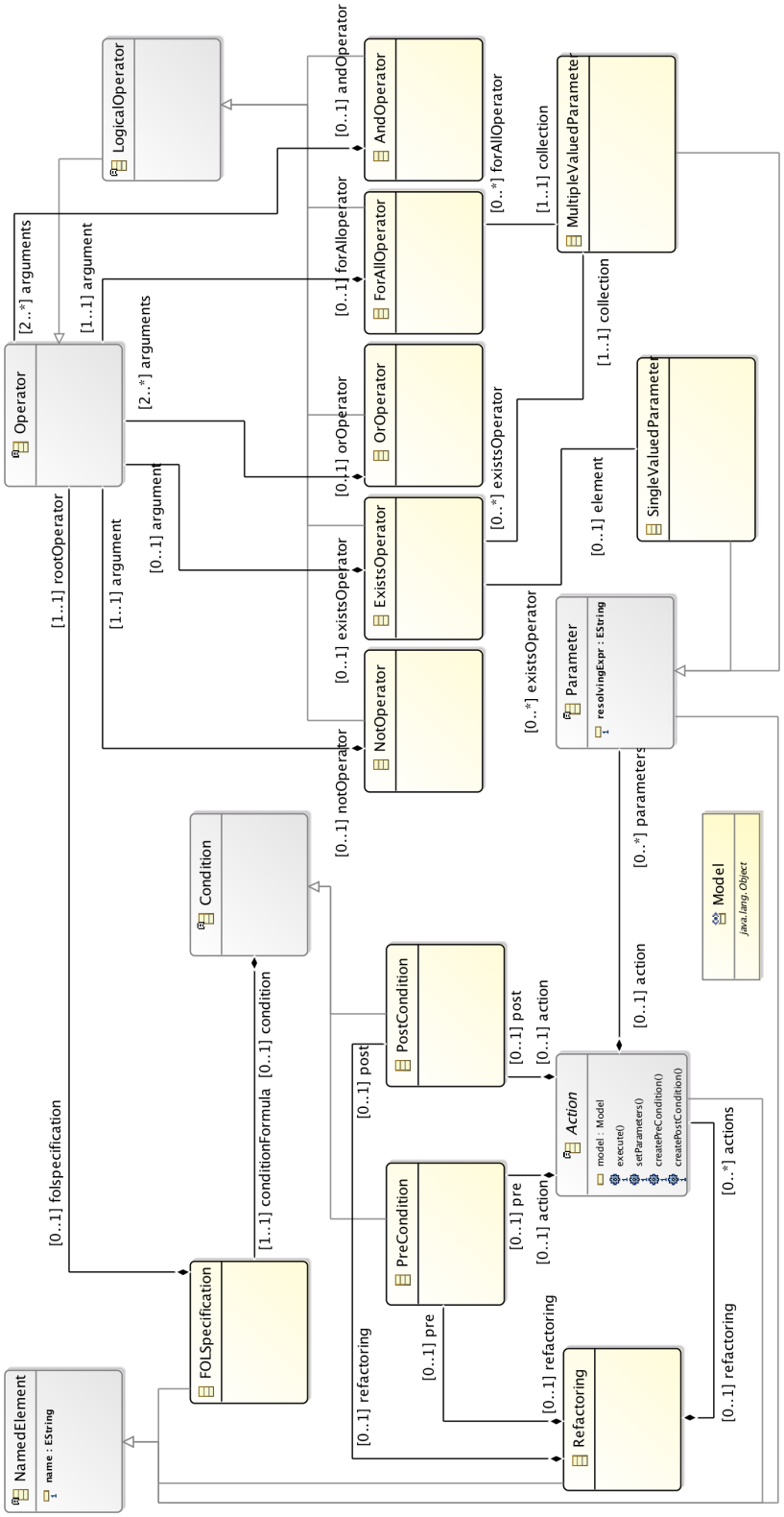

Figure 1: The metamodel for model refactoring

- $post_1 = (\exists\ c \in C) \wedge (\forall\ t \in T\ \exists\ t \in N) \wedge (\forall\ d \in D\ \exists\ d \in T)$, where C and N are the set of all the `Components` and `Nodes` in the UML model, respectively, whereas D is the set of deployment `Nodes` of c, after the application of *moveComp*. Hence, $post_1$ is similar to pre_1, but it additionally checks that the moved `Component` is actually deployed to the selected target `Nodes`.
- $pre_2 = \exists\ c \in C$, where C is the set of all the `Components` in the UML model. Hence, pre_2 checks the existence of `Component` to be deleted.
- $post_2 = \exists\ c \in C$, where C is the set of all the `Components` in the UML model, after the application of *moveComp*. Hence, $post_2$ checks the non-existence of the deleted `Component`.

Given the refactoring actions above and their pre- and post-conditions, the sequence $moveComp(c1, T1) \rightarrow deleteComp(c1)$ would be feasible, because $post_1$ (applied to the instances *c1* and *T1*) does not break pre_2 (applied to *c1*), i.e. *c1* still exists after having been moved and it can be thus deleted. Conversely, the sequence $deleteComp(c1) \rightarrow moveComp(c1, T1)$ would not be feasible, because $post_1$ (applied to *c1*) breaks pre_2 (applied to *c1* and *T1*), i.e. *c1* does not exist anymore after having been deleted, hence it cannot be moved.

Providing a language-independent automated support for a priori verifying the feasibility of refactoring action sequences, through the mechanism exemplified above, fills an important lack in the context of model-based refactoring, which is often conceived as a user-driven activity, i.e. semi-automated. In fact, such an automated support paves the way to approaches that automatically generate (and possibly suggest) design alternatives resulting from the application of different refactoring action sequences, e.g. search-based approaches based on multi-objective optimization such as [2, 15, 21].

An `Action` exposes four public methods, which must be implemented to exploit the framework for a specific modelling language ([2]): i) *execute*, ii) *setParameters*, iii) *createPreCondition* and iv) *createPostCondition*. The latter two methods allow to define the pre- and post- conditions that must be validated before (i.e. pre-) and after (i.e. post-) the action is executed, by invoking the method *execute*. However, in order to create a `Condition` for an `Action`, its `Parameters` have to be set by overriding the *setParameters* method. Parameters are used to "resolve" model elements involved in Conditions, through the *resolvingExpr* strings that are assumed to be conform to the OCL grammar (i.e. they must express OCL queries), as described later in this section. There are two types of Parameter: `SingleValued` and `MultipleValued`. The former type is used to resolve a single model element, whereas the latter is used for resolving a set of model elements. `Parameters` may properly recur within a `FOLSpecification` as operands of the `Operators` that compose the specification. In fact, a `FOLSpecification` contains a root, namely `rootOperator`, which represents the first `Operator` that, in turn, may contain other `Operators`, and so on. As a result, a `FOLSpecification` is expressed in a functional-like notation by using the available kinds of `Operators`. Currently, `LogicalOperators` have been defined, as: `Exists`, `ForAll`, `Not`, `And`, `Or`. However, the metamodel definition is open to the introduction of other operators, if needed, such as operational ones.

Note that, different representations of `FOLSpecification` would have been possible, e.g. as a set of atomic expressions which are firstly declared and then composed by means of operators. However, for sake of simplicity, we have opted for the one presented here.

Equation (1) describes a minimal example of `FOLSpecification`, where the existence of a specific model element, resolved through the *getCompToMoveSVP* `SingleValuedParameter`, has to be verified within the set of all the UML `Components`, which is retrieved via the *getAllCompsMVP* `MultipleValuedParameter`. Practically, the `FOLSpecification` will be composed by only one `ExistsOperator` with `getCompToMoveSVP` as *element* and `getAllCompsMVP` as *collection*.

$$\exists getCompToMoveSVP \in getAllCompsMVP \tag{1}$$

3.2 Main Facilities for Language-Specific Contexts

As mentioned before, our metamodel is an instrument that allows to generate a set of refactoring support facilities. In this section, we describe what a user is expected to do in order to exploit such refactoring facilities with the aim of "plugging" the metamodel into a specific modelling notation (e.g. UML [19]).

We remark that `Action` represents a key anchor point, because a user must implement her refactoring actions, by overriding the four declared `Action` methods, in order to apply the metamodel to a specific modeling language. As from Fig. 2, the metamodel provides an ad-hoc class, namely `Manager`, that exposes all the facilities needed for such purpose.

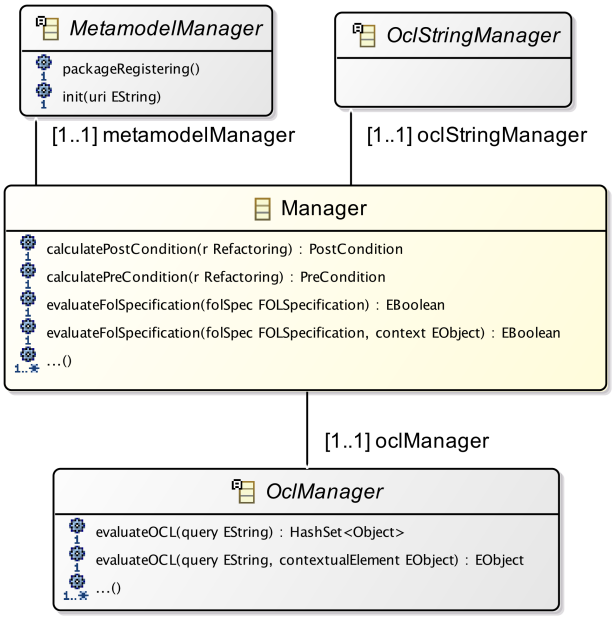

Figure 2: Excerpt of `Manager` metaclass and its facilities.

[2]Note that we do not provide any specific concrete `Action` within the metamodel, because it would jeopardize the generic nature of our framework. A categorization of refactoring actions, such as additions, deletions and changes would be possible, but this would have introduced a further degree of complexity.

3.2.1 Manager Facilities. `Manager` provides many methods, spanning from objects initialization and creation (e.g. of `FOLSpecifications`, `Single` and `MultipleValuedParameters`, `Pre` and `PostConditions`, `Operators`) to conditions comparison, processing and evaluation. Fig. 2 depicts a minimal but significant subset of such methods, which are described in the following to clarify the fundamental role played by `Manager`:

- *calculatePreCondition* is in charge of deriving the `PreCondition` of the `Refactoring`, passed as input to the method, by exploiting an ad-hoc version of the approach by Ó Cinnéide et al. [17] that we have implemented for our purposes.
- *calculatePostCondition* is in charge of deriving the `PostCondition` of the `Refactoring`, passed as input to the method, analogously to *calculatePreCondition*.
- *evaluateFolSpecification* verifies a `FOLSpecification` against a contextual model element passed as input parameter, which has to be retrieved and properly cast, based on the target modelling language.

Moreover, `Manager` has three fundamental references to different kinds of managers, aimed at tailoring to a specific modelling language, namely: `MetamodelManager`, `OclManager` and `OclStringManager`. To this aim, a user has to extend and implement her own version of the latter managers, as described in the following.

3.2.2 MetamodelManager. `MetamodelManager` handles ordinary activities on a target modelling language. Two methods need to be overridden by a user extension of this class, i.e. *init* and *packageRegistering*. The former has to initialise all paths of the artefacts needed for dealing with models conforming to the target metamodel. For example, the path of the source model onto which a `Refactoring` has to be applied, and thus pre- and post- conditions have to be derived and verified, as well as additional resources and resource sets ([3]) have to be defined in the *init* method. The *packageRegistering* method, instead, is responsible for the registration of all the necessary packages into the EMF registry, and it has to be properly called by *init* to enable the creation of required resources.

3.2.3 OclManager. `OclManager` is in charge of performing OCL queries and handling their results, within the context of the target modeling language. Notice that OCL query results return Objects, HashSets of Objects, or Java Generics (not shown in Fig. 2), which need to be properly cast before being returned to the caller. For this reason, the user is required to extend `OclManager` with her own manager, by providing tailored implementations of its predeclared methods.

3.2.4 OclStringManager. As mentioned before, parameters are used to resolve model elements involved within conditions. Although this appears as a restricting assumption, it simplifies the management of such strings. As alternative, in fact, OCLExpressions ([4]) rather than strings should be used, thus resulting in a higher complexity. However, in both cases, the user is required to know OCL, hence we have opted for introducing such assumption in favour of simplicity.

That said, `OCLStringManager` itself does not provide any method, hence the user has to implement all the necessary OCL queries within a tailored extension of it. Each method has to return a predefined query, and all queries will be available to use by *setParameters* methods of the defined refactoring actions. When the `FOLSpecification` of a `Condition` for an `Action` has to be evaluated against a model, all the needed `Parameters` are set by construction, thus each OCL query can be evaluated to properly resolve the model elements involved in the condition.

Finally, we remark that the basic idea behind such mechanism for resolving model elements, grounded on a single class that provides all the needed strings, is borrowed from the Android string resources system, which can be queried in order to obtain text strings with optional text styling and formatting for the labels of an application ([5]).

4 APPLYING THE METAMODEL TO AN EXAMPLE MODELING LANGUAGE : UML

In this section, we illustrate the application of our metamodel to a specific modeling language. We have recently applied refactoring action sequences automatically, in two different model-based contexts, that are: i) an Epsilon-based approach, where different kinds of refactoring sessions on UML-MARTE models [18], aimed at detecting and removing performance antipattern occurrences, can be executed by exploiting the different execution semantics of Epsilon languages [2]; ii) a multi-objective optimization approach, where alternative sequences of refactoring actions on Æmilia architectural specifications [7, 9] are automatically obtained as the near-Pareto front of a fitness function that considers design aspects (i.e., number of changes) and performance ones (i.e. number of performance antipattern occurrences and a performance quality indicator) [3]. For this reason, in this section we opt for providing a thorough usage example of our metamodel facilities within one of these contexts, i.e UML-MARTE, rather than showing the effects of refactoring actions on modeling examples. However, readers interested to this aspect can refer to the works mentioned above.

4.1 UmlAction

Let us consider a refactoring `Action`, namely `UmlMoveComponent`, which takes as input a UML `Component`, namely *compToMove*, and a list of `Nodes`, namely *targets*, and deploys the former to the latter. As mentioned before, the Java implementation of the action has to be introduced within the *execute* method.

While implementing an action, a user has to define attributes representing its `Parameters` that will be subsequently set within the *setParameters* method. Listing 1 shows an excerpt of an implementation of the *setParameters* method conceived for the `UmlMoveComponent` refactoring action. In particular, it shows how to set the following `Parameters`:

- *compToMoveSVP*, whose *resolvingExp* retrieves the `Component` to move (line 2). The *resolvingExpr* is set through the *getCompQuery* method.

[3] The terms *resource* and *resource set* here refer to the classes org.eclipse.emf.ecore.resource.Resource and org.eclipse.emf.ecore.resource.ResourceSet, respectively.
[4] From org.eclipse.ocl.expressions.
[5] https://developer.android.com/guide/topics/resources/string-resource.

Listing 1: Excerpt of *setParameters* method for UmlMoveComponent

```
1  public void setParameters() {
2   setCompToMoveSVP(Manager.getInstance().
      createSingleValuedParameter(
      UmlOclStringManager.getInstance().
      getCompQuery(compToMove)));
3   setAllCompsMVP(Manager.getInstance().
      createMultipleValuedParameter(
      UmlOclStringManager.getInstance().
      getAllCompsQuery()));
4   setTargetNodesMVP(...getNodesQuery(nodes)));
5   setAllNodesMVP(...getAllNodesQuery()));
6   ...
7  }
```

- *allCompsMVP*, whose *resolvingExpr* retrieves all the Components within the system model (line 3). The *resolvingExpr* is set through the *getAllCompsQuery* method.
- *targetNodesMVP*, whose *resolvingExpr* retrieves all the Nodes to which *compToMove* has to be moved (line 4). The *resolvingExpr* is set through the *getNodesQuery* method.
- *allNodesMVP*, whose *resolvingExpr* retrieves all the Nodes within the system model (line 5). The *resolvingExpr* is set through the *getAllNodesQuery* method.

The *resolvingExpr*s above must be provided by a properly tailored OclStringManager, namely UmlOclStringManager (lines 4–15 and 16–18 of Listing 6).

Once action parameters have been set, they can be retrieved through the proper getters, e.g. *getTargetNodesMVP()* and *getAllNodesMVP()*, in order to define action pre- and post- conditions. Hence, using such a notation, the PreCondition of UmlMoveComponent should verify the following first-order logics formula:

$$\exists\, getCompToMoveSVP() \in getAllCompsMVP() \wedge \forall t \in getTargetNodesMVP()\ \exists\, t' \in getAllNodesMVP() \quad (2)$$

This pre-condition checks the existence of:

(1) the Component to move (i.e. *getCompToMoveSVP()*) within the set of all the Components (i.e. *getAllCompsMVP()*);
(2) each target deployment Node for the Component to move (i.e. *getTargetNodesMVP()*) within the set of all the Nodes (i.e. *getAllNodesMVP()*).

Listing 2 shows a possible implementation of the formula (2). A PreCondition is firstly created (line 2), which will be finally set as the precondition of UmlMoveComponent (line 13). A FOLSpecification for the PreCondition is then created (line 3), which will be set as the *conditionFormula* of the previously created PreCondition (line 12). Thereafter, an AndOperator is created (line 4), which represents the *rootOperator* of the previously created FOLSpecification (line 11). The first argument of the *rootOperator* (added at line 6) is an ExistsOperator, namely *existsCompToMove* (line 5), whose element to search is the UML Component to move. The latter is targeted by *getCompToMoveSVP()* and has to be searched within the set of all Components, which are targeted by *getAllCompsMVP()*. The second argument of the *rootOperator*

Listing 2: *createPreCondition* method for UmlMoveComponent Action

```
1  public void createPreCondition() {
2   PreCondition pre =
      LogicalSpecificationFactory.eINSTANCE.
      createPreCondition();
3   FOLSpecification fol = Manager.getInstance().
      createFOLSpecification("MoveCompPreC");
4   AndOperator folRoot = Manager.getInstance().
      createAndOperator();
5   ExistsOperator existsComp = Manager.
      getInstance().createExistsOperator(
      getCompToMoveSVP(), getAllCompsMVP());
6   folRoot.getArguments().add(existsComp);
7   ForAllOperator forallTargets = Manager.
      getInstance().createForAllOperator(
      getTargetNodesMVP());
8   ExistsOperator existsInNodes = Manager.
      getInstance().createExistsOperator(
      getAllNodesMVP());
9   forall.setArgument(existsInNodes);
10  folRoot.getArguments().add(forallTargets);
11  fol.setRootOperator(folRoot);
12  preCondition.setConditionFormula(fol);
13  setPre(pre);
14 }
```

(added at line 10) consists of a ForAllOperator, namely *forallTargets*, whose *collection* targets the *targetNodesMVP* of the refactoring action through the proper getter (line 7). The argument of such ForAllOperator is an ExistsOperator, namely *existsInNodes*, whose *collection* points to the *allNodesMVP* of the refactoring action through the proper getter (lines 8–9).

The PostCondition of UmlMoveComponent should verify the following first-order logics formula:

$$\exists\, getCompToMoveSVP() \in getAllCompsMVP() \wedge \forall t \in getTargetNodesMVP()\ \exists\, t' \in getAllNodesMVP() \wedge \forall n \in getDeployNodesMVP()\ \exists\, n' \in getTargetNodesMVP() \quad (3)$$

This post-condition checks the existence of:

(1) the Component to Move (i.e. *getCompToMoveSVP()*) within the set of all the Components (i.e. *getAllCompsMVP()*);
(2) each target deployment Node for the Component to move (i.e. *getTargetNodesMVP()*) within the set of all the Nodes (i.e. *getAllNodesMVP()*);
(3) each Node onto which the Component has to be moved (i.e. *getDeployNodesMVP()*) within the set of all target deployment Nodes (i.e. *getTargetNodesMVP()*)

Listing 3 shows a possible implementation of formula (3). From an implementative point of view, building a PostCondition is like building a PreCondition. For this reason, we only report an excerpt of *createPostCondition()* for UmlMoveComponent refactoring action. Such excerpt shows an implementation of the third predicate of Equation (3). Practically, it is implemented through a ForAllOperator, namely *forallDeployNodes* (line 3), with a nested

Listing 3: *createPostCondition* method for UmlMoveComponent Action

```
1 public void createPostCondition() {
2  ...
3  ForAllOperator forallDeployNodes = Manager.
     getInstance().createForAllOperator(
     getDeployNodesMVP());
4  ExistsOperator existsInTargetNodes = Manager.
     getInstance().createExistsOperator(
     getTargetNodesMVP());
5  forallDeployNodes.setArgument(
     existsInTargetNodes);
6  ...
7 }
```

Listing 4: *init* method for a UmlMetamodelManager

```
1 public void init(String modelUri) {
2  setModelUri(modelUri);
3  packageRegistering();
4  getUmlOclManager().inizialize(getResourceSet
     ());
5  setUmlResource((UMLResource) getResourceSet()
     .getResource(Manager.getInstance().
     string2FileUri(getModelUri()), true));
6  model = (Model) EcoreUtil.getObjectByType(
     getUmlResource().getContents(), UMLPackage.
     Literals.MODEL);
7 }
```

`ExistsOperator`, namely *existsInTargetNodes* (lines 4–5), aimed at verifying that, after the refactoring action execution, all the `Nodes` onto which *compToMove* is deployed are within the set of target `Nodes` (lines 4–5). This is achieved by exploiting the `MultipleValuedParameters`, namely *deployNodesMVP* and *targetNodesMVP*.

4.2 UmlMetamodelManager

As said in Sec. 3.2.2, a `MetamodelManager` has to handle the ordinary activities on a target modelling language. Two methods have to be overridden by a potential `UmlMetamodelManager`, namely *init* and *packageRegistering*. An example of the former is reported in Listing 4 (lines 1–7), where the model URI is set (line 2), all the needed packages are registered (line 3), the `UmlOclManager` is initialized (line 4) and, finally, resources are properly set (line 5) and the model is stored in ad-hoc variable (line 6). Note that the code not shown here, i.e. *packageRegistering()* method, works on UML models profiled with MARTE that are out of the scope of this paper.

4.3 UmlOclManager

As said in Sec. 3.2.3, an `OclManager` has to handle OCL queries execution and their results. Methods of `OclManager` have to be implemented by a potential `UmlOclManager`. For sake of paper readability, we do not report here a complete example, however a minimal excerpt for an *evaluateOCL* method is reported in Listing 5, where an OCL object and an OCLHelper ([6]) are instantiated (lines

[6]From org.eclipse.ocl.helper.

Listing 5: Example of *evaluateOCL* method for a UmlOclManager

```
1 public Object evaluateOCL(String query, Object
    contextualElement) throws ParserException {
2  OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?,
     Constraint, EClass, EObject> ocl = OCL.
     newInstance(EcoreEnvironmentFactory.
     INSTANCE);
3  OCLHelper<EClassifier, ?, ?, Constraint>
     helper = ocl.createOCLHelper();
4  if (contextualElement instanceof Model)
5   helper.setContext(UMLPackage.Literals.MODEL)
      ;
6  else
7   ...
8  Query<EClassifier, EClass, EObject> q = ocl.
     createQuery(helper.createQuery(query));
9  return q.evaluate(contextualElement);
10 }
```

2–3), the query context is set by properly checking the type of the contextual element (lines 4–7) and, furthermore, an OCL Query to evaluate is properly obtained (line 8). The latter is finally evaluated and the result of such evaluation is returned (lines 9–10). Note that, the code not shown here (line 7), performs further checks of the contextual element type (e.g., `Component`, `Node`) in order to properly set the helper context.

4.4 UmlOclStringManager

As said in Sec. 3.2.4, `OCLStringManager` itself does not provide any method, hence the user has to implement all the necessary OCL queries, for example within a `UmlOclStringManager`. Each method has to return a predefined query, and all queries will be available to the *setParameters* method of the `UmlMoveComponent` refactoring action. Listing 6 reports sample code for some methods of a `UmlOclStringManager`, in particular:

- *getCompQuery* (lines 1–3), which retrieves the `Component` to move, passed as input, if the PaRunTInstance stereotype (from PAM package of MARTE profile) is applied.
- *getAllCompsQuery* (lines 4–6), which retrieves all the `Components` with the PaRunTInstance stereotype applied.
- *getNodesQuery* (lines 7–17), which retrieves all the `Nodes` with the GaExecHost stereotype (from GQAM package of MARTE profile) applied, among the ones passed as input.
- *getAllNodesQuery* (lines 18–20), which retrieves all the `Nodes` with the GaExecHost stereotype applied.

5 CONCLUSION AND FUTURE WORK

In this paper, we have presented a metamodel for the specification and verification of model refactoring action sequences. The metamodel contains constructs related to the model-based refactoring domain, such as the concepts of action, pre- and post- condition, and a number of logical operators that can be used for defining first-order logical formulae representing pre- and post- conditions of single refactoring actions.

Listing 6: Sample methods for a UmlOclStringManager

```
1  public String getCompQuery(Component c) {
2   return "Component.allInstances()->select(c1 |
       c1.getAppliedStereotypes()->exists(s | s.
     name = 'PaRunTInstance'))->select(c2 | c2.
     name = '" + c.getName() + "' )->asSequence
     ()->first()";
3  }
4  public String getAllCompsQuery() {
5   return "Component.allInstances()->select(c |
     c.getAppliedStereotypes()->exists(s | s.
     name = 'PaRunTInstance'))";
6  }
7  public String getNodesQuery(List<Node> nl) {
8   String query;
9   query = "Node.allInstances()->select(node |
     node.getAppliedStereotypes()->exists(s | s.
     name = 'GaExecHost'))->" + "select(n | ";
10  Iterator<Node> iterator = nl.iterator();
11  while (iterator.hasNext()) {
12   query += "n.name = '" + iterator.next().
      getName() + "'";
13   if (iterator.hasNext())
14    query += " or ";
15  }
16  return query + " )";
17 }
18 public String getAllNodesQuery() {
19  return "Node.allInstances()->select(node |
     node.getAppliedStereotypes()->exists(s | s.
     name = 'GaExecHost'))";
20 }
```

The metamodel has been conceived to be plugged into different modeling languages (e.g. UML) through a number of facilities supporting objects creation, as well as the definition of language-specific managers. It also embeds the implementation of an already existing mechanism for deriving pre- and post- conditions of refactoring action sequences [17].

A critical point is that the user is in charge of writing her pre- and post-condition as OCL statements, thus she needs to acquire confidence with OCL that, however, is one of the most widely adopted languages in the MDE environment.

As mentioned in Sec. 4, we have recently introduced automation in the application of refactoring action sequences in two model-based contexts, i.e. UML and Æmilia. Here, we have illustrated the application of our metamodel to the UML case.

Our primary objective in the near future is to extend this metamodel to enable the specification of performance antipatterns detection rules and their verification against a model. To this aim, we shall need to include relational operators for comparing numerical values retrieved through the resolving expressions of single- and multiple- valued parameters. Relational operators would allow our metamodel to potentially provide a complete support to the specification and verification of first-order logics formulae, independently from the fact that they represent pre- and post- conditions or performance antipatterns detection rules.

REFERENCES

[1] R. Abílio, J. Padilha, E. Figueiredo, and H. Costa. 2015. Detecting Code Smells in Software Product Lines – An Exploratory Study. In *12th International Conference on Information Technology - New Generations, ITNG*. IEEE, 433–438.

[2] Davide Arcelli, Vittorio Cortellessa, Mattia D'Emidio, and Daniele Di Pompeo. 2018. EASIER: an Evolutionary Approach for multi-objective Software archItecturE Refactoring. In *2018 IEEE International Conference on Software Architecture, ICSA*. IEEE, 105–114.

[3] Davide Arcelli, Vittorio Cortellessa, and Daniele Di Pompeo. 2018. Performance-driven software model refactoring. *Journal for Information and software Technology* 95 (2018), 366–397.

[4] Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. 2012. Antipattern-based model refactoring for software performance improvement. In *8th International Conference on the Quality of Software-Architectures, QoSA*. ACM, 33–42.

[5] Thorsten Arendt and Gabriele Taentzer. 2013. A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering* 20, 2 (June 2013), 141–184.

[6] Jagdish Bansiya and Carl G Davis. 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering* 28, 1 (2002), 4–17.

[7] Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. 2002. Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. In *Performance Evaluation of Complex Systems Techniques and Tools*. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 236–260.

[8] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. 2009. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 271–285.

[9] Martina De Sanctis, Catia Trubiani, Vittorio Cortellessa, Antinisca Di Marco, and Mirko Flamminj. 2017. A model-driven approach to catch performance antipatterns in ADL specifications. *Journal for Information and software Technology* 83 (2017), 35–54.

[10] Martin Fowler, Kent Beck, John Brant, William F Opdyke, and Don Roberts. [n. d.]. *Refactoring: improving the design of existing code*. Addison-Wesley.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

[12] Adnane Ghannem, Ghizlane El-Boussaidi, and Marouane Kessentini. 2013. Model refactoring using interactive genetic algorithm. In *5th International Symposium on Search Based Software Engineering*. Springer Berlin Heidelberg, 96–110.

[13] Jongwook Kim, Don Batory, and Danny Dig. 2015. Scripting parametric refactorings in Java to retrofit design patterns. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 211–220.

[14] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. 2016. Improving Refactoring Speed by 10X. In *38th International Conference on Software Engineering (ICSE)*. ACM, 1145–1156.

[15] Usman Mansoor, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. 2015. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal* (2015), 1–29.

[16] Iman Hemati Moghadam and Mel Ó Cinnéide. 2011. Code-Imp: A Tool for Automated Search-based Refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools (WRT)*. ACM, 41–44.

[17] Mel Ó Cinnéide and Paddy Nixon. 2000. Composite refactorings for Java programs. In *Workshop on Formal Techniques for Java Programs (FTfJP)*.

[18] OMG. 2009. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. https://www.omg.org/omgmarte/

[19] OMG. 2017. Unified Modeling Language (UML 2.5.1). http://www.omg.org/spec/UML/index.htm

[20] William F Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation.

[21] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Mel Ó Cinnéide, Kalyanmoy Deb, and Katsuro Inoue. 2015. A Multi-Objective Refactoring Approach to Introduce Design Patterns and Fix Anti-Patterns. In *North American Search Based Software Engineering Symposium (NASBaSE)*. 1–16.

[22] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modelling Framework*. Addison-Wesley Professional.

[23] Wuliang Sun, Robert B. France, and Indrakshi Ray. 2013. Analyzing Behavioral Refactoring of Class Models. In *Workshop on Models and Evolution co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (ME@MoDELS 2013) (CEUR Workshop Proceedings)*. CEUR-WS.org, 70–79.

[24] Michael Wahler, Uwe Drofenik, and Will Snipes. 2016. Improving Code Maintainability: A Case Study on the Impact of Refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 493–501.