Specification, Execution, and Detection of Refactorings for Software Models

Philip Langer, Konrad Wieland, and Petra Brosch
Business Informatics Group
Institute for Software Systems and Interactive Systems
Vienna University of Technology, Austria
{langer,wieland,brosch}@big.tuwien.ac.at

ABSTRACT

Predefined automatically applicable *composite operations* such as *refactorings* are a prerequisite for *efficient software modeling*. Some modeling environments provide an initial set of basic refactorings, but they hardly offer extension points for user-specified refactorings. Even if extension points exist, the introduction of new refactorings requires programming skills and deep knowledge of the respective metamodel of the used modeling language.

We present EMF Modeling Operations, a Java™ based framework for specifying and executing composite operations within the user's modeling language and editor of choice. The user demonstrates a composite operation on a concrete example from which a generic and executable operation specification is semi-automatically derived. Furthermore, we show how the resulting specification may be used to enable an a-posteriori detection of applications of the specified operations between two successive versions of a model, also in absence of a directly recorded change log.

1 Introduction

With the rise of *model-driven development*, software models are lifted to first-class artifacts in software development. Like in the traditional code-oriented software development, software models are iteratively refined and, therefore, heavily evolve during their life cycle. During development, *recurring composite operations* such as *refactorings* are applied on software artifacts to enhance the readability, maintainability and extensibility. Consequently, for code artifacts several approaches have been realized in practice to support the *automatic execution* of refactorings on existing code artifacts. However, for software models such techniques are rare [1]. Although some modeling environments provide a basic set of executable refactorings, they hardly offer any extension points for adding user-specified refactorings. Even if extension points exist, the introduction of new custom refactorings requires programming skills and deep knowledge of the respective metamodel. As a consequence, the specification of new refactorings is only practicable for experienced programmers.

To open the specification to *users without programming skills*, we present EMF Modeling Operations, a Java™ based framework enabling the development of executable

composite operation specifications. Comparable to macro recording in Microsoft® Office products, a new specification is created by *demonstrating the composite operation*, fine-tuning the automatically derived operation's pre- and postconditions, and, if necessary, adding additional augmentations such as iterations.

Once a model operation is specified, it may be recurrently applied to arbitrary models using the Operation Execution Engine. This engine enables a time-saving repetition of recurring refactoring in modeling environments. Moreover, to help developers retrospectively understanding a model's evolution, applications of specified model operations, applied between two successive versions of an evolving model, may be *detected a posteriori* using the Operation Detection Engine.

EMF Modeling Operations is realized as an Eclipse plug-in and may be used for any EMF-based models[1]. In the following, we outline the functioning of each component, in particular, the Operation Recorder in Section 2, the Operation Execution Engine in Section 3, and the Operation Detection Engine in Section 4.

2 Operation Specification By Demonstration

Operations such as refactorings may be described by a set of atomic operations, namely, *create*, *update*, *delete*, and *move* which are executed on a model adhering to specific preconditions [2]. Furthermore, to allow for complex attribute value computations in the target model as well as to enable the detection of occurrences of the specified composite operation in generic change scripts (cf. Section 4), also postconditions are included in the operation specification.

A direct way to realize *operation specification by demonstration* is to record each user interaction within the modeling environment as proposed in [3] for programming languages and in [4] for models. However, this would demand an intervention in the modeling environment, and due to the multitude of modeling environments, we refrain from this possibility. Instead, we apply a *state-based comparison* to determine the executed operations after modeling the initial model and the final model. This allows the use of any editor without depending on editor-specific change recording. To overcome the imprecision of heuristic state-based approaches, a unique ID is automatically assigned to each model element before the user illustrates the changes. Moreover, the Operation Recorder is designed in such a way to be independent from any specific modeling language, as long as it is based on EMF Ecore or the metametamodel is mapped to Ecore.

Following our design rationale, we propose a two-phase *operation demonstration process* which is supported by the Operation Recorder (cf. Fig. 1). For a detailed description of the underlying approach, we kindly refer to [5].

Phase 1: Modeling. First, the user creates the *initial model* containing all essential model elements to apply the composite operation. Next, each element of the initial model is automatically annotated with an ID, and a so-called *working model*, i.e., a copy of the initial model for demonstrating the composite operation, is created. The IDs preserve the relationship of the original elements in the initial model and the changed elements in the working model. Finally, the user performs the complete composite operation on the working model in her familiar modeling environment by applying all necessary atomic operations. The output of this step is the *revised model*, which is together with the initial model the input for the second phase of the operation specification process.

Phase 2: Configuration & Generation. Due to the unique IDs of the model elements, the atomic operations of the operation may precisely be determined automat-

[1] `http://www.eclipse.org/emf`

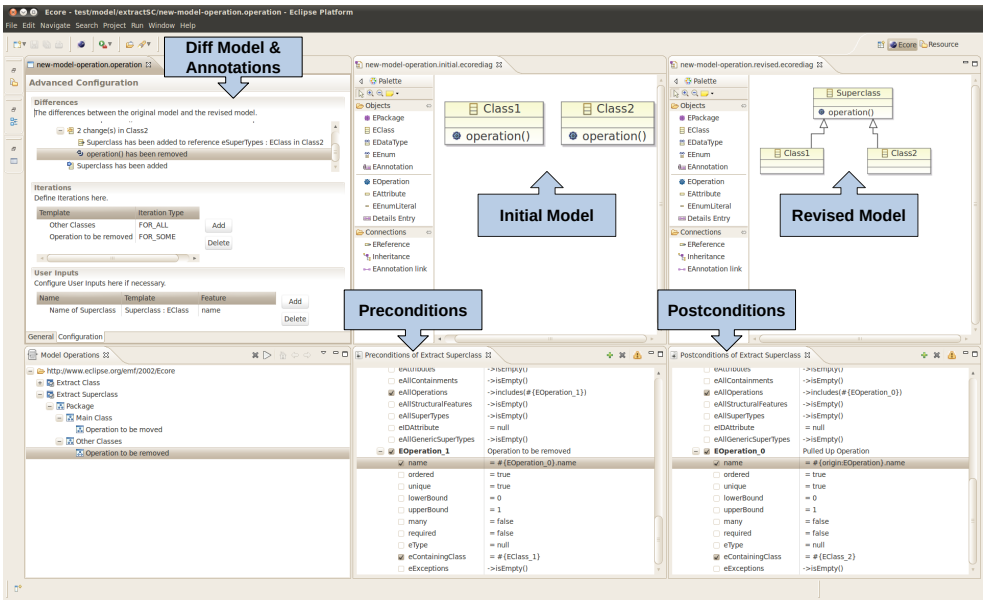

Figure 1: Screenshot of the EMF Model Macro Editor.

ically using a state-based comparison. The results are saved in a *diff model* containing all detected atomic changes. Subsequently, an initial version of the *pre-* and *postconditions* (cf. lower right view in Fig. 1) of the operation is inferred by analyzing the initial model and revised model, respectively. Sometimes, the automatically inferred conditions do not completely express the intended pre- and postconditions of the operation. Thus, they only act as a basis for accelerating the operation specification process and may be refined by the user. In particular, conditions may be relaxed, enforced, and modified and further annotations such as *iterations and user inputs* may be specified (cf. upper left view in Fig. 1). Finally, the *Operation Specification Model* is generated, which is a self-contained and complete description of the specified operation consisting of the initial and revised model, the diff model, and the pre- and postconditions.

3 Execution of Operation Specifications

Once the Operation Specification Model is created, it may be applied to (parts of) arbitrary models fulfilling to the operation's preconditions. To start the execution, the user selects a model element in an arbitrary model and provides a so-called *prebinding* by linking the selected model element to the corresponding model element in the operation specification's example model. With this, the selected model element in the arbitrary model is specified to be transformed equally to the linked model element in the operation specification. Based on the operation's preconditions, the Operation Execution Engine automatically completes the provided prebinding and presents it—if a valid and complete binding was found—to the user as depicted in Fig. 2a. Next, the Operation Execution Engine queries the user for additionally configured user inputs and finally performs the operation.

To realize the Operation Execution Engine, we faced two major challenges. First, the condition evaluation engine has to cope with cyclic condition dependencies. There-

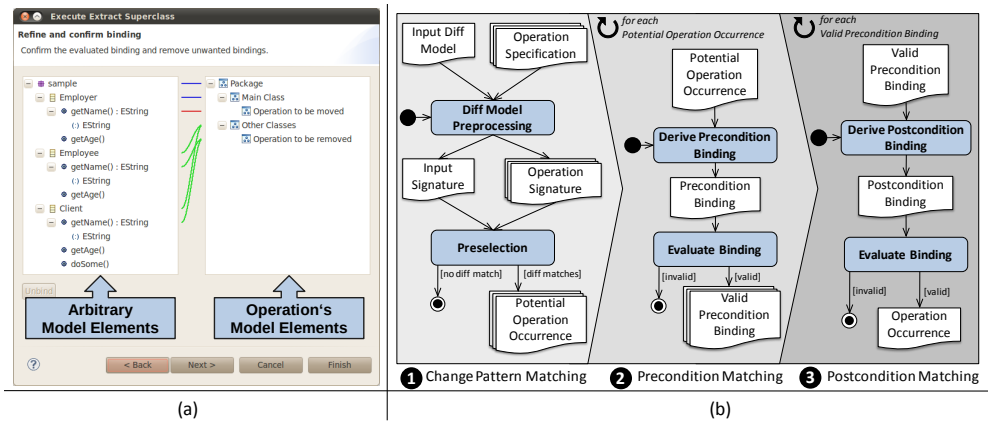

Figure 2: (a) Screenshot of the execution binding dialog. (b) Detection process.

fore, we use a backtracking algorithm to explore all potentially valid combinations of model element bindings based on the user-specified prebinding to finally find a complete and valid binding. Second, the repeated execution of the detected atomic changes poses some challenging issues. For instance, an added element might refer to already existing model elements. Consequently, new elements may not simply be copied to the target, otherwise the copy would still link to elements contained by the operation's example model. Instead, they have to be copied and accordingly *rewired* to ensure that the new model element only refers to existing elements in the currently transformed model.

4 A-Posteriori Detection of Operation Applications

Given two successively modified versions of one model, we now aim to detect applications of the defined operations. The detection process (cf. Fig. 2b) takes as input a difference report (*input diff model*) obtained by a state-based comparison of two successively modified model versions as well as the list of detectable operation specifications. The process consists of three phases: *(i)* the *preselection* of potential operation occurrences is accomplished by searching for the change patterns of the provided operation specifications in the input diff model. Subsequently, for each potential operation occurrence, *(ii)* the preconditions are evaluated, and finally, *(iii)* the postconditions are checked. If both are valid, an application of an operation is detected.

Change Pattern Matching. The goal of this phase is to enable an efficient and fast triage of operation occurrence candidates that potentially may have been applied according to the input diff model. To allow for a fast search, the diff models (input diff model and the operation specification's diff models) are translated into easily processable signatures. For each provided operation specification we now check whether its signature is contained in the input signature. If a match is found, the respective operation has potentially been applied and we proceed with the condition matching in the following phases.

Precondition Matching. For each potential operation occurrence, the preconditions of the respective operation specification are evaluated. For this, a binding of affected model elements to the operation specification's *initial model elements* has to be derived (*precondition binding* in Fig. 2b) which is then evaluated using the aforementioned condition evaluation engine (cf. Section 3). If one operation specification

has been applied more than once, the evaluation engine returns a valid precondition binding for each potential occurrence. This list of valid precondition bindings serves as input for the next phase.

Postcondition Matching. For each valid precondition binding, a *postcondition binding*, i.e., a binding of changed model elements to the corresponding operation specification's *revised model elements* is derived and evaluated. If a valid postcondition binding is found, an occurrence of an operation application is at hand.

5 Conclusions and Ongoing Work

In this paper, we outlined EMF Modeling Operations, a Java™ based framework for specifying and recurrently executing operations. With the Operation Detection Engine, this framework allows to retrospectively detect applications of user-specified operations. EMF Modeling Operations will soon be available from our project homepage[2] using the open source license EPL[3].

Ongoing work comprises *optional changes* allowing to specify parts of the operation to be optional. Furthermore, we currently elaborate on *computing the inverse* of an operation specification as well as its *composition* to enable the creation of new (composite) composite operations from two or more existing ones. Finally, we plan to set up a *community server* allowing to easily exchange existing operations.

References

[1] Tom Mens, Michel Wermelinger, Stephane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in Software Evolution. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22. IEEE Computer Society, 2005.

[2] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In *Model-driven Software Development—Research and Practice in Software Engineering*, pages 199–217. Springer, 2005.

[3] Romain Robbes and Michele Lanza. Example-Based Program Transformation. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, pages 174–188. Springer, 2008.

[4] Yu Sun, Jules White, and Jeff Gray. Model transformation by demonstration. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, pages 712–726. Springer, 2009.

[5] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, pages 271–285. Springer, 2009.

[2] `http://www.modelversioning.org`
[3] `http://www.eclipse.org/legal/epl-v10.html`