The Operation Recorder: Specifying Model Refactorings By-Example [*]

Petra Brosch [†] Martina Seidl
Konrad Wieland Manuel Wimmer

Business Informatics Group
Vienna University of Technology, Austria
lastname@big.tuwien.ac.at

Philip Langer

Department of Telecooperation
Johannes Kepler University Linz, Austria
philip.langer@jku.at

Abstract

Predefined composite operations are handy for efficient software modeling, e.g., for the automatic execution of refactorings, and for the introduction of patterns in existing models. Some modeling environments provide an initial set of basic refactoring operations, but hardly offer any extension points for the user. Even if extension points exist, the introduction of new composite operations requires programming skills and deep knowledge of the respective metamodel.
In our demonstration we present the Operation Recorder, a tool for specifying composite operations, like refactorings, within the user's modeling language and environment of choice. The user models the composite operation by-example, which enables the semi-automatic derivation of a generic composite operation specification. This specification may be used in further modeling scenarios, like model refactoring and model versioning. We demonstrate our tool by creating two refactoring specifications for UML class diagrams and UML state machine diagrams.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software—Reuse models

General Terms Design, Languages

Keywords refactoring, composite operation, by-example approach

[*] This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-819584.

[†] Funding for this research was provided by the fFORTE WIT - Women in Technology Program of the Vienna University of Technology, and the Austrian Federal Ministry of Science and Research.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

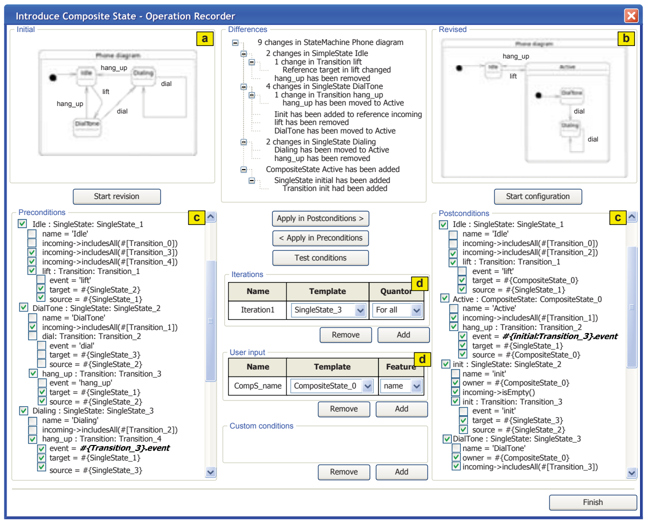
1. Introduction

With the rise of model-driven development software models are lifted to first-class artifacts in the software development lifecycle. Like in the traditional code-oriented software development process, software models are iteratively refined and restructured. In this software development phase recurrent (sequences of) operations, also known as refactorings, are applied to ensure the readability, maintainability and extensibility. A well established approach for specifying and communicating a recurrent sequence of operations is to give it a name and define a pattern, as is done, e.g., by [Gamma et al. 1995] and by [Fowler et al. 1999].

So far, the technical specification of these refactorings has to be performed by experts, because they require extensive programming effort and deep knowledge of the development environment. To open the specification of refactorings to modelers, we present the Operation Recorder, an approach for the user-friendly modeling of composite operations by-example.

The Operation Recorder enables the specification of composite operations by modeling concrete examples at the model layer, i.e., the same layer the operation specification is applied. It allows the user to create these examples within her preferred modeling language and editor of choice, and then to automatically derives a generic operation specification from these example refactorings.

The benefit of the resulting operation specification is twofold. On the one hand, our Execution Engine is able to apply operation specifications on arbitrary models containing a pattern matching the initial model. On the other hand, the Detection Engine detects occurrences of specified operations in generic model differences obtained by a state-based comparison. In combination with the Execution Engine, the recognition and replay of applied refactorings also improve model versioning [Dig et al. 2006].



```
9 changes in StateMachine Phone diagram
  2 changes in SimpleState Idle
    1 change in Transition lift
      Reference target in lift changed
    hang_up has been removed
  4 changes in SingleState DialTone
    1 change in Transition hang_up
      hang_up has been moved to Active
    Iinit has been added to reference incoming
    lift has been removed
    DialTone has been moved to Active
  2 changes in SingleState Dialing
    Dialing has been moved to Active
    hang_up has been removed
  CompositeState Active has been added
    SingleState initial has been added
      Transition init had been added
```

```
☑ Idle : SingleState: SingleState_1
  ☐ name = 'Idle'
  ☐ incoming->includesAll(#[Transition_0])
  ☑ incoming->includesAll(#[Transition_3])
  ☑ incoming->includesAll(#[Transition_4])
  ☑ lift : Transition: Transition_1
    ☐ event = 'lift'
    ☑ target = #{SingleState_2}
    ☑ source = #{SingleState_1}
☑ DialTone : SingleState: SingleState_2
  ☐ name = 'DialTone'
  ☑ incoming->includesAll(#[Transition_1])
  ☐ dial: Transition: Transition_2
    ☐ event = 'dial'
    ☐ target = #{SingleState_3}
    ☐ source = #{SingleState_2}
  ☑ hang_up : Transition: Transition_3
    ☐ event = 'hang_up'
    ☑ target = #{SingleState_1}
    ☑ source = #{SingleState_2}
☑ Dialing : SingleState: SingleState_3
  ☐ name = 'Dialing'
  ☐ incoming->includesAll(#[Transition_2])
  ☑ hang_up : Transition: Transition_4
    ☑ event = #{Transition_3}.event
    ☑ target = #{SingleState_1}
    ☑ source = #{SingleState_3}
```

Name	Template	Quantor
Iteration1	SingleState_3	For all

Name	Template	Feature
CompS_name	CompositeState_0	name

```
☑ Idle : SingleState: SingleState_1
  ☐ name = 'Idle'
  ☐ incoming->includesAll(#[Transition_0])
  ☑ incoming->includesAll(#[Transition_2])
  ☑ lift : Transition: Transition_1
    ☐ event = 'lift'
    ☑ target = #{CompositeState_0}
    ☑ source = #{SingleState_1}
☑ Active : CompositeState: CompositeState_0
  ☐ name = 'Active'
  ☑ incoming->includesAll(#[Transition_1])
  ☑ hang_up : Transition: Transition_2
    ☑ event = #{initial:Transition_3}.event
    ☑ target = #{SingleState_1}
    ☑ source = #{CompositeState_0}
☑ init : SingleState: SingleState_2
  ☐ name = 'init'
  ☑ owner = #{CompositeState_0}
  ☑ incoming->isEmpty()
  ☑ init : Transition: Transition_3
    ☐ event = 'init'
    ☑ target = #{SingleState_3}
    ☑ source = #{SingleState_2}
☑ DialTone : SingleState: SingleState_3
  ☐ name = 'DialTone'
  ☑ owner = #{CompositeState_0}
  ☑ incoming->includesAll(#[Transition_3])
```

Figure 1. Screenshot of the Operation Recorder.

2. Operation Definition

To create new composite operation specifications the user provides some general information and models the initial situation (cf. Fig. 1a) in her preferred modeling environment. This initial model is automatically annotated with unique IDs and copied by the Operation Recorder. Thus, the modeler performs all operations the composite operation consists of on this working copy, again in her preferred modeling environment (cf. Fig. 1b). When the modeler confirms the revised working copy, the Operation Recorder precisely detects all performed operations by conducting a state-based comparison relying on the ID-based match. Furthermore, the Operation Recorder derives pre- and postconditions from the initial and the revised working model (cf. Fig. 1c). For each model element and its features a condition is generated. Since the generic condition generation facility is not able to decide which feature is essential to act as pre- and postcondition, the modeler may relax, enforce, and modify the automatically derived conditions by deactivating, reactivating, and changing them. If necessary for the respective composite operation, the modeler may additionally define iterations and user input variables (cf. Fig. 1d). The Operation Recorder is realized as an Eclipse plug-in and may be used for any EMF-based models. To detect all operations performed on the example model we apply EMF Compare. For a more detailed description of the Operation Recorder we kindly refer to the AMOR project homepage[1] and [Brosch et al. 2009a].

As already mentioned, operation specifications are applicable on arbitrary models using the Execution Engine. If a part of a model matches the preconditions the derived differences which consist of atomic operations are interpreted and applied on the matching model elements. This enables a time-saving repetition of recurring complex operations in modeling environments like the macro recording feature in Microsoft Excel. Furthermore, operation specifications are self-contained, so they may be centralized and easily shared with other modelers over a certain platform, e.g., a community server. The detection of occurrences of composite operations in the context of model versioning allows a more compact representation of the difference reports by folding atomic operations which belong to a composite operation. Thus, detecting applied composite operations enables a faster and better understanding of the modeler's original intention. Furthermore, it enables a smarter conflict detection and resolution as proposed in [Dig et al. 2006]. We implemented the Detection Engine by searching for the operation pattern contained in the operation specification. If the pattern is found and the model elements referenced by the matching operations fulfill the pre- and postconditions an occurrence of the composite operation is at hand.

[1] http://www.modelversioning.org

3. Outline of the Demonstration

After outlining the approach from a theoretical point of view we will demonstrate the easy and efficient usage of our tool by creating two refactoring specification in the domain of the UML class diagram as well as the UML state machine diagram. For educational reasons we start with a small example, the *Convert to Singleton* refactoring, where a plain UML class is transformed to act as singleton. To get a complete impression of the expressive power we continue with a more complicated example, *Introduce Composite State* in UML state machine diagrams, for which the definition of user input as well as iterations are necessary. For both of these refactorings we show how these are applied in arbitrary models and demonstrate how occurrences of these can henceforth be detected to improve model versioning [Dig et al. 2006].

References

Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. AMOR - Towards Adaptable Model Versioning. In *MCCM'08 @ MoDELS'08*, 2008.

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. *Accepted for MoDELS'09*, 2009a.

Danny Dig, Tien N. Nguyen, Kashif Manzoor, and Ralph Johnson. MolhadoRef: A Refactoring-aware Software Configuration Management Tool. In *OOPSLA'06*. ACM, 2006.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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