An Example is Worth a Thousand Words: Composite Operation Modeling By-Example[⋆]

Petra Brosch[1⋆⋆], Philip Langer[2], Martina Seidl[1], Konrad Wieland[1], Manuel Wimmer[1], Gerti Kappel[1], Werner Retschitzegger[3], and Wieland Schwinger[2]

[1] Business Informatics Group, Vienna University of Technology, Austria
`{lastname}@big.tuwien.ac.at`
[2] Department of Telecooperation, Johannes Kepler University Linz, Austria
`{firstname.lastname}@jku.ac.at`
[3] Information System Group, Johannes Kepler University Linz, Austria
`werner@ifs.uni-linz.ac.at`

Abstract. Predefined composite operations are handy for efficient 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 this paper, we introduce a method for specifying composite operations 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 implemented the approach in the Operation Recorder and performed an evaluation by defining multiple complex refactorings for UML diagrams.

Key words: refactoring, composite operation, by-example approach

1 Introduction

Since modeling is hardly done in terms of single atomic operations but by performing a sequence of operations to reach a desired goal, 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. [11] and Fowler et al. [10]. In order to define patterns not only for human interaction, but also in a machine readable and executable format, composite operations may be described as model transformations.

[⋆] 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.

So far, the implementation of such model transformations has mainly been accomplished by experts, because they require extensive programming effort and deep knowledge of APIs of modeling environments, metamodels, and dedicated transformation languages. The contribution of this paper is to overcome this pitfall. To open the specification of patterns and refactorings to modelers, we present the Operation Recorder, a front-end 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 pattern definition is applied. It allows the user to work within her preferred modeling language and editor of choice, without leaving the familiar environment. The examples consist of the initial model, the revised model, and the differences between them. These differences of the two models are then generalized by the Operation Recorder and may be applied to arbitrary models containing a pattern matching the initial model. In the research project AMOR [2], we use the Operation Recorder in two orthogonal modeling settings, namely for refactoring and versioning.

Refactoring. Predefined refactoring operations as known from IDEs like Eclipse[4] find their way into modeling environments. Since it is not possible to provide all refactorings out-of-the-box—this is especially the case if domain specific modeling languages (DSMLs) are employed—the modeling editor should offer extension points for editing and adding user-defined refactorings [16]. The Operation Recorder is used as user-friendly front-end for specifying user-defined composite operations.

Versioning. The state-based recognition of multiple atomic operations as one refactoring may also improve model versioning [6]. Since refactorings often have global effects in the overall model, subsuming a set of atomic changes to only one change makes it easier to read version histories and to understand model evolution. In the case of optimistic versioning, where parallel editing of model artifacts is allowed, the recognition of refactorings improves automatic merge as discussed in [9]. Even if an automatic merge cannot be performed, manual conflict resolution is accelerated by providing a more readable conflict report. A comprehensive model versioning environment is part of our future research.

The paper is organized as follows. Starting with a motivating example in Section 2 we outline the process of composite operation modeling by-example in Section 3. Section 4 provides a detailed account of the implementation of the Operation Recorder and Section 5 summarizes the evaluation results for complex refactorings for UML diagrams. Section 6 discusses related work and we conclude with an outlook on our future work in Section 7.

2 Motivating Example

To emphasize our motivation for developing the Operation Recorder, we discuss the refactoring "Introduce Composite State" for UML statecharts. The concrete

[4] http://www.eclipse.org/

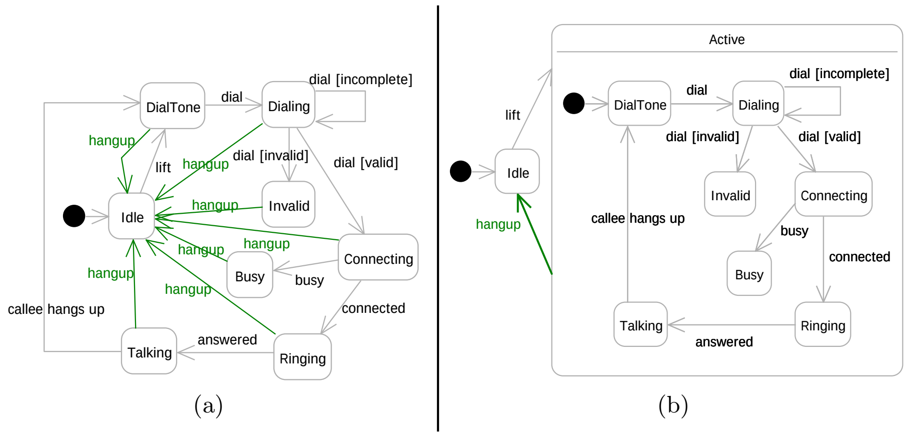


Fig. 1. (a) Initial Phone Statechart. (b) Refactored Phone Statechart. [20]

example illustrating the possible states and transitions of a phone conversation was taken from Sunyé et al. [20] (cf. Figure 1).

Whenever a hangup event occurs in the unrefactored model shown in Figure 1(a), the phone moves to the state *Idle*. The multitude of similar transitions, which are pointing to the state *Idle* and which are triggerd by the same event, allows the application of the refactoring pattern “Introduce Composite State”, i.e., introducing a composite state and folding the hangup transitions to one single transition as depicted in Figure 1(b). The modification consists of the following changes:

1. A composite state named *Active* is created.
2. All states except *Idle* are moved into *Active*.
3. The outgoing hangup transitions of these states are folded to one single transition which leaves the composite state *Active*.
4. The transition *lift* is split to an incoming transition of *Active* and to the initial pseudostate of *Active*.

In most modeling tools, the general specification of such a composite operation is only possible by an implementation in a textual programming language, which demands dedicated programming skills. Based on our experience when developing the “Introduce Composite State” refactoring in Java, the solution comprises nearly 100 lines of code implementing the pure refactoring logic, not counting preconditions on the applicability of the refactoring pattern and code realizing a front-end for the application of the refactoring pattern by the user.

Another alternative to specify composite operations is the use of dedicated model transformation languages. This enables the development of composite operations in a more compact form by, e.g., applying declarative rule-based languages. In this way, single transformation steps may be described declaratively by transformation rules. However, specifying a set of transformation rules and their interactions is currently supported by a few transformation engines only, and requires a deep understanding of the transformation process. Furthermore, model transformation approaches are rarely included in current modeling environments. Thus, tool adapters are required to use these technologies and the users have to switch to a new environment, which again calls for dedicated knowledge.

Modelers, as the potential users of the composite operation specification facilities are familiar with the notation, semantics, and pragmatics of the modeling languages they use in daily business. They are not experts, however, in programming languages, transformation techniques, as well as specific APIs.

With the Operation Recorder we aim at providing a tool, which makes the specification of composite operations practicable to every modeler.

3 By-Example Operation Specification at a Glance

Composite operations may be described by a set of atomic operations, i.e., *create*, *update*, *delete*, and *move* which are executed on a model in a specific modeling scenario, i.e., adhering to specific preconditions [24]. Furthermore, to enable the detection of occurrences of the specified composite operation in generic change scripts, we need to include also postconditions to the composite operation specification.

An immediate way to realize composite operation specification by-example is to record each user interaction within the modeling environment as proposed in [18] for programming languages. 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 modification 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 Ecore [7].

Following our design rationale, we propose a two-phase by-example operation specification process as shown in Figure 2. In the following, we discuss this two-phase specification process step-by-step.

Phase 1: Modeling. In the first step, the user models the initial situation in her familiar modeling environment, i.e., the model required in order to apply the composite operation. The output of this step is called *initial model.* In the second step, 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 by applying changes, is created. The IDs preserve the relationship of the original elements in the initial model and changed elements in the revised model. Doing so, we are able to precisely detect all atomic changes, i.e., also element moves. Consequently, the generated match between the initial model and the revised model is sound and complete. In the third step, the user performs the complete composite operation on the working model, again 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.

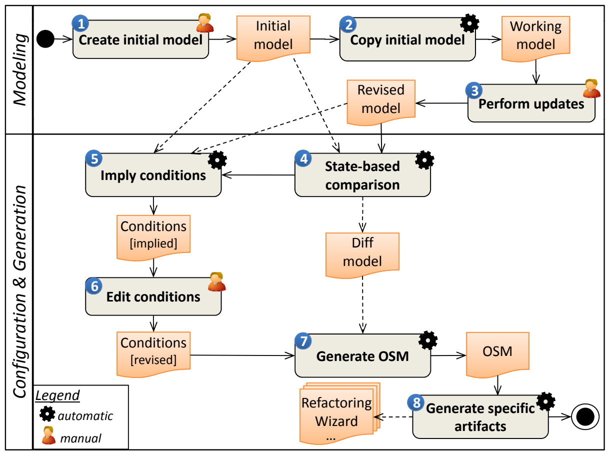
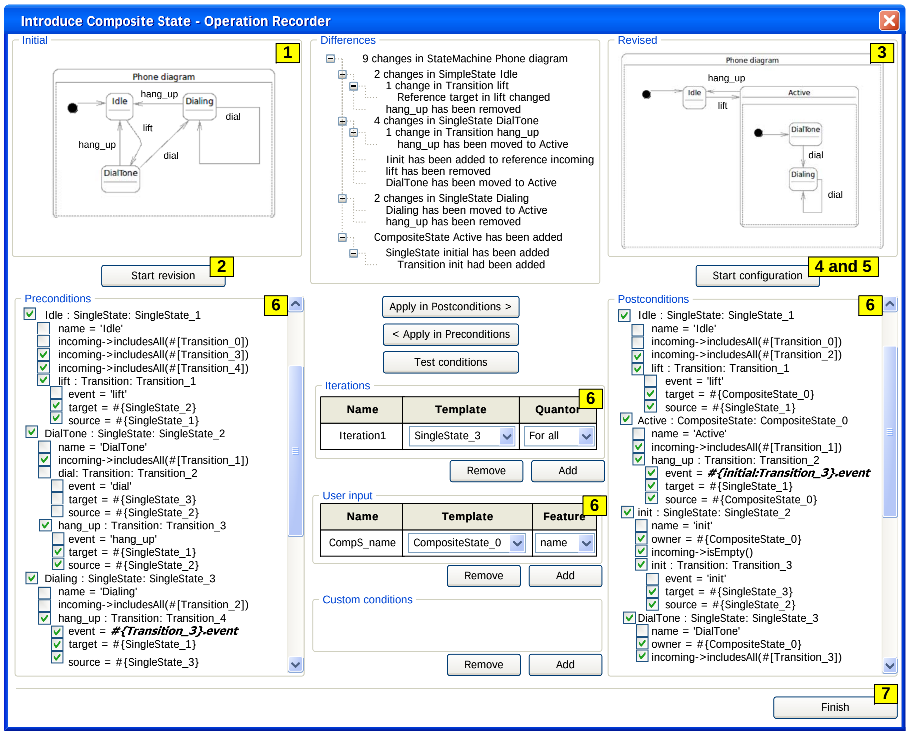


Fig. 2. By-Example Operation Specification Process

Phase 2: Configuration & Generation. Due to the unique identifiers of the model elements the atomic operations of the composite operation may be determined automatically in Step 4 using a state-based comparison. The results are saved in the *diff model*. Subsequently, an initial version of *pre-* and *postconditions* of the composite operation is inferred in Step 5 by analyzing the initial model and revised model, respectively. Usually, the automatically generated conditions from the example are too strong and do not express the intended pre- and postconditions of the composite operation. They only act as a basis for accelerating the operation specification process and have to be refined by the user in Step 6. In particular, parts of the conditions may be activated and deactivated within a dedicated environment with one mouse click. Generated conditions may be modified by the user and additional conditions may be added. After the configuration of the conditions, the *Operation Specification Model* (OSM) is generated in Step 7, which consists of the diff model and the revised pre- and postconditions. Finally, from the OSM, specific artifacts may be generated in Step 8 such as refactoring wizards which allow the automatic execution of refactorings. Another use case of the OSM would be to directly act as a template for change scripts for finding the applications of composite operations between different model versions.



Differences
- 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

Preconditions
```
✓ 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}
```

Apply in Postconditions >

< Apply in Preconditions

Test conditions

Iterations

Name	Template	Quantor
Iteration1	SingleState_3	For all

Remove Add

User input

Name	Template	Feature
CompS_name	CompositeState_0	name

Remove Add

Custom conditions

Remove Add

Postconditions
```
✓ 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])
```

Fig. 3. Front-end of the Operation Recorder

4 By-Example Operation Recorder in Action

In the previous section we illustrated the operation specification process from a generic point of view. In the following, we define the refactoring "Introduce Composite State" from Section 2 showing the operation specification process from the user's point of view. The specification process is supported by the 8-step Operation Recorder using the front-end depicted in Figure 3 (for details on the implementation of the front-end cf. [1].

Step 1: Create initial model. The modeler starts with modeling the initial situation in the upper left area labeled *Initial* (cf. Figure 3). For this task, the modeler may apply any editor of her choice, since the Operation Recorder is independent of editor-specific operation tracking, using a solely state-based comparison. In this step every model element, which is necessary to show the composite operation, has to be introduced. It is not necessary to draw every state of the diagram shown in Figure 1(a). Therefore, in the *Initial* area only those states are depicted, which will later on be modified differently. The first of those is the state *Idle*, which will remain outside the composite state we will add later. Second, the state *DialTone*, which will be moved to the newly added composite state acting as first state and finally the state *Dialing*, which only will be moved to the composite state loosing its transition to *Idle*. There is no need to model e.g. the state *Connecting* shown in Figure 1(a), since it is equally modified like *Dialing*. For such states, which have to be handled equally, the Operation Recorder provides techniques to define iterations in the later configuration phase.

Step 2: Copy initial model. When the modeler finishes the initial model she confirms it by pushing the button *Start editing*. This initiates the automatic copy process which adds a unique ID to every model element of the initial model before the working copy is created.

Step 3: Perform updates. After the ID-annotated working copy is created, it is displayed in the upper right area of the front-end, named *Revised*. Now, the modeler performs each operation the composite operation consists of on the revised model. In our example, the modeler has to add a composite state named *Active*, move the single state *DialTone* and *Dialing* into it, introduce a new initial state in *Active*, connect it with *DialTone* and change or remove the other transitions. As soon as the composite operation is completely executed, the modeler finalizes the modeling phase by pushing the *Start configuration* button.

Step 4: State-based comparison. In this step, the comparison between the initial model and the revised model is done to automatically identify the previously executed changes. When the comparison is completed, the detected differences show up in the upper center area named *Differences*.

Step 5: Imply conditions. Next, the Operation Recorder automatically implies the preconditions from the initial model and the postconditions from the revised model. The condition generation process for the pre- and the postconditions is similar. For each model element in the respective model, a so-called *template* is created. A template describes the role, a model element plays in the specific composite operation. When executing or detecting a defined composite operation, concrete model elements are evaluated against and subsequently bound to these templates. In the front-end the pre- and postconditions are illustrated on the lower left and lower right area, respectively. Each template contains conditions displayed beside the template names. Each of the automatically generated condition constrains the value of a specific feature. In our example, the area *Preconditions* shows three different templates in the first level for the model elements *Idle*, *DialTone*, and *Dialing* and their respective preconditions. These templates have a user-changeable symbolic name, e.g., *SingleState_1*, and are arranged in a tree to indicate their containment relationships. Templates may also be used as a variable in condition bodies to generically express a reference to other model elements or their values. We use the syntax `#{Transition_3}.event` to access the *event* property of the first element matching the template *Transition_3*. To reference *all* matching elements in a conditions body the syntax `#[Template_name]` is used. The scope of a template is either the initial model or the revised model. Nevertheless it is possible to access the template of the opposite model in the conditions using the prefixes `initial:` and `revised:`, respectively.

Step 6: Edit conditions. Usually, the conditions automatically generated in the previous step are too strong and do not express the intended pre- and postconditions of the composite operation. They only act as a kickstart accelerating the operation specification process and have to be manually refined in this step. The Operation Recorder offers three different instruments to adapt the generated conditions.

First, the modeler may *relax* or *enforce* conditions. This is simply done by activating or deactivating the checkboxes in front of the respective templates or conditions. If a template is relaxed all containing conditions are deactivated. By default, conditions constraining *string* features and `null`-values are deactivated, as in our experience they are not relevant in most of the cases. In the running example, four templates and three conditions in preconditions as well as three templates and two conditions in the postcondition have to be relaxed, additionally to the by-default deactivated conditions. For instance, in the preconditions the templates representing the initial state and implicitly its contained transition as well as the template representing the reflexive transition *dial* in state *Dialing* are not relevant and have to be relaxed properly. The same is true for the condition `incoming->includesAll(#{Transition_0})` in template *Idle* as it is not necessary that this state has the incoming transition matching *Transition_0*.

Second, the modeler may *modify* conditions by directly editing them. For our example it is necessary to specify that a state which is moved into the composite state has to own the event which is folded as outgoing transition (in our example *hang_up*). For this reason, the condition in the preconditions and the postconditions highlighted in bold font are modified to express this constraint.

Finally, users may adapt the composite operation specification by *augmenting*, e.g., defining iterations and annotating necessary user input for setting parameters of the composite operations. In our example, the modeler has to introduce one iteration for the template *SingleState_3*. This iteration specifies that the two operations executed on this template have to be repeated *for all* its matching model elements. In other words, defining this iteration, all model elements containing the transition to be folded are moved to the composite state. Further, the modeler introduces a user input for the property *name* of template *CompositeState_0* to indicate a value which has to be set by the user of the refactoring. Apparently, iterations may only be specified for templates from the initial model and user input for features of templates from the revised model. To ensure the syntactic and semantic correctness of all conditions, the modeler may test all conditions against the initial or revised model by pushing the *Test conditions* button. A failing condition indicates a wrongly specified constraint, because the conditions have to match at least the example models.

Step 7: Generate OSM. To finalize the operation specification, the modeler pushes the *Finish* button. This initiates the generation of the OSM. This model contains all necessary information for further usage like its detection of occurrences in generic difference models or its execution in various specific models. Operation specifications are conform to the metamodel depicted in Figure 4. The class `CompositeOperationSpecification` contains general information about the operation like the name, a description as well as the initial and revised model, the pre- and postconditions, the iterations, and the differences. For the initial and the revised model kept in the attributes `initialModel` and

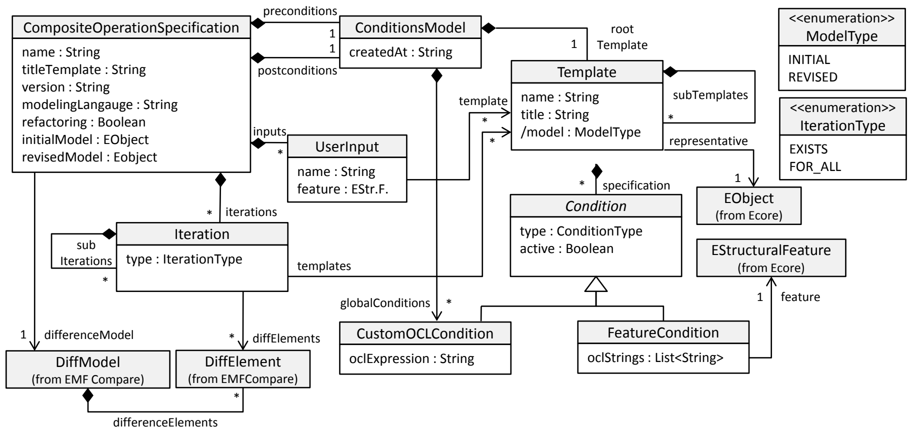


Fig. 4. Operation Specification Metamodel

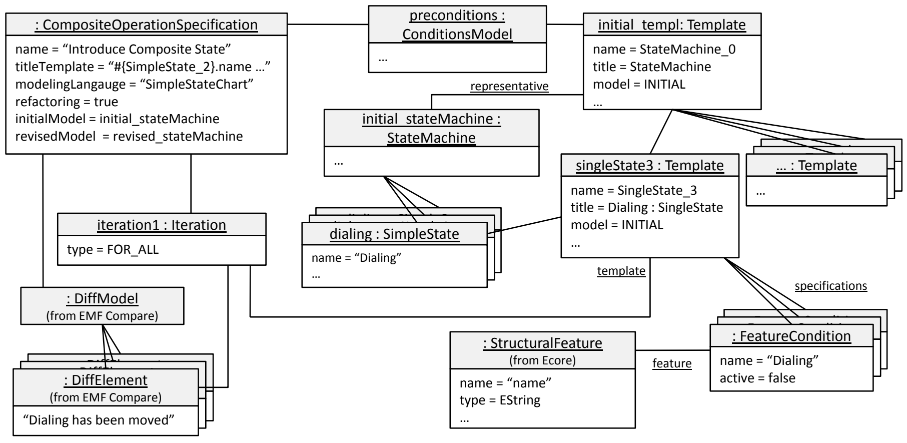

: CompositeOperationSpecification
name = "Introduce Composite State"
titleTemplate = "#{SimpleState_2}.name …"
modelingLangauge = "SimpleStateChart"
refactoring = true
initialModel = initial_stateMachine
revisedModel = revised_stateMachine
preconditions : ConditionsModel
initial_templ: Template
name = StateMachine_0
title = StateMachine
model = INITIAL
representative
initial_stateMachine : StateMachine
singleState3 : Template
name = SingleState_3
title = Dialing : SingleState
model = INITIAL
… : Template
dialing : SimpleState
name = "Dialing"
iteration1 : Iteration
type = FOR_ALL
template
specifications
: DiffModel (from EMF Compare)
: DiffElement (from EMF Compare)
"Dialing has been moved"
: StructuralFeature (from Ecore)
name = "name"
type = EString
feature
: FeatureCondition
name = "Dialing"
active = false

Fig. 5. Excerpt of the Operation Specification Model for the Running Example

`revisedModel`, the class `CompositeOperationSpecification` holds a reference to `ConditionsModel` which consists of a root `Template` representing the previously mentioned root object of the initial or revised model's conditions. Each template may have a number of subtemplates corresponding to the containment hierarchy of the elements in the initial or revised model. The specific model element in the initial or revised model is referenced in `representative`. Furthermore, a `Template` is specified by a list of custom conditions and feature conditions. `FeatureConditions` constrain the value of a specific feature and are generated automatically in Step 5.

Figure 5 illustrates an excerpt of the object diagram representing the OSM for the previously described statechart example. This diagram highlights some aspects, like the introduced iteration, the template hierarchy and its references to the concrete model elements as well as an instance of a `FeatureCondition`

Listing 1.1. Generated OCL code

```
...
attr singleState_1      : SingleState = ... /* selected by user */
attr transition_3       : Transition = ... /* selected by user */
...
self.includesAll(self.outgoingTransition->select(
    event = transition_3.event and target = singleState_1
    and source = self))
```

Listing 1.2. Generated Refactoring Code

```
method introduceCompositeState(String cs_name,
        Transition transition_3, SingleState singleState_1){
    ...
    //Create composite state
    CompositeState cs = new CompositeState(cs_name);
    ...

    //Shift States into composite state
    Iterator iter = states.select(s|cond(s)).iterator();
    while (iter.hasNext()){
        State state = iter.hasNext();
        cs.ownedStates().add(state);
        state.outgoing().remove(state.outgoing.select(t|
            t.event = transition_3.event and
            t.target = singleState_1);
        ...
    }

    ... //Create additional elements and link them properly
}
```

for the feature *name*. All of these components have their counterpart in the the front-end already presented in Figure 3.

Step 8: Generate specific artifacts. The last step of the process is the generation of specific artifacts out of the OSM for the utilization outside the Operation Recorder. To execute the refactoring specification depicted in Figure 3, the user has to choose which single state remains outside the composite state and which single state should be transformed to the starting node within the composite state. This is done by simply binding the respective single states to the templates *SingleState_1* and *SingleState_2*. To keep this process user-friendly, users do not have to bind model elements based on template names directly. Instead, users are referred to the example initial model and have to assign concrete elements to the elements of this example. Then, the direct binding to the templates is induced automatically. Based on this binding the concrete transition element matching *Transition_3* is evaluated. As an example for a generated artifact, in Listing 1.1 the OCL code evaluating all model elements for which the iteration has to be applied (i.e., which match the template *SingleState_3*) is illustrated. The code implementing the iteration itself is shown in Listing 1.2.

5 Evaluation

For evaluating the effort necessary to define new composite operations with the Operation Recorder we performed a case study with the objective to specify five refactorings for the UML class diagram and two refactorings for the UML statechart (cf. Table 1). Those well-known refactorings were adapted—if necessary—for the application on models as those refactorings are mostly defined for the application on code. The complexity of the refactorings varies from simple, e.g., "Move Attribute", to complex, e.g., "Introduce Composite State". Due to space limitations, we kindly refer to our project page for a detailed description [1].

The values shown in Table 1 reflect the effort for the user to specify the refactorings. The *#Template* column refers to the number of templates derived from the initial model and for the revised model, respectively, in order to establish the pre- and postconditions. The *#Conditions/#Selected* column contains the total number of pre- and postconditions as well as the number of initially selected conditions. These numbers are strongly related to the size of the metamodel employed for the editor. In our experiments, we used specialized UML editors which allowed us to focus on efficiently testing the refactorings. For example, if we had used the full UML2 editor for "Move Attribute" we would have obtained again 4 templates, but more than 100 conditions. To increase readability, we plan to integrate general filters in the Operation Recorder as well as to provide extension points for metamodel specific filters, which allow to hide unused metamodel features.

The column *#Diffs* shows the number of differences between the initial model and the revised model. The concrete value depends on the way a refactoring is modeled. We asked for example two modelers to specify the "Introduce Composite State" refactoring starting from the same initial model. Although their revised models contained the same elements, the one performed 9 changes whereas the other needed 14 changes. The first reused the existing elements of the model and modified them accordingly, whereas the other deleted them and introduced new elements.

The configuration effort is reflected by the remaining columns of Table 1. The *#Relax/#Enforce* column describes how many conditions have to be (un)selected manually by the user. *#Modifications* refers to the number of edits which have to be performed and the last column shows the number of introduced iterations. In general, our case study showed, that the configuration effort mostly consists of relaxing conditions which is done with some clicks. The few condition modifications are typically needed to refer to properties of other templates and therefore are easily accomplished. Even for more complicated refactorings, e.g. "Extract Superclass", only a few configuration steps are necessary.

Overall, the Operation Recorder approach allowed a very intuitive specification of the refactorings where the tasks which have to be performed by a human user are straightforward. In future work, we plan to perform a more extensive evaluation with a wide range of modelers with different levels of modeling experiences.

Refactoring	#Templates		#Conditions/ #Selected		#Diffs	#Relax / #Enforce		#Modifications		#Iterations
	Pre	Post	Pre	Post		Pre	Post	Pre	Post	
mvAtt	4	4	13/3	13/3	1	3/0	3/1	0	1	0
convSing	3	4	8/2	11/4	2	2/2	2/2	1	1	0
encVar	3	7	8/3	16/8	4	3/0	2/4	0	3	0
repDV	3	5	8/3	16/6	4	3/0	2/2	0	1	0
extSC	5	5	16/4	18/6	6	0/2	0/0	1	2	1
intCS	11	13	41/27	48/32	9	7/0	5/0	1	1	1
merge	10	8	36/22	29/17	6	6/0	4/0	2	0	1

Table 1. Refactorings: Move Attribute (mvAtt), Convert to Singleton (convSing), Encapsulate Variable (encVar), Replace Data Value with Object (repDV), Extract Superclass (extSC), Introduce Composite State (intCS), Merge States (merge)

6 Related Work

In this section, we give a short overview of work related to our by-example operation specification approach organized in the categories *composite operations for models* and *model transformation by-example.*

Composite operations for models. Most existing approaches for defining composite operations focus solely on model refactorings. One of the first investigations in this area was done by Sunyé et al. [20] who define a set of UML refactorings on the conceptual level by expressing pre- and post-conditions in OCL. Boger et al. [5] present a refactoring browser for UML supporting the automatic execution of pre-defined UML refactorings within a UML modeling tool. While these two approaches only focus on pre-defined refactorings, approaches by Porres [17], Zhang et al. [24], Kolovos et al. [13], and Verbaere et al. [22] allow the introduction of user-defined refactorings in dedicated textual programming languages. A similar idea is followed by Mens [15] and Biermann et al. [4] who use graph transformations to describe the refactorings within the abstract syntax of the modeling languages. The application of this formalism comes with the additional benefit of formal analysis possibilities of dependencies between different refactorings. In any case, the definition of new refactorings requires intense knowledge of the modeling language's metamodel, of special APIs to process the models, and finally of dedicated programming language. In other words, very specific expertise is demanded.

The Operation Recorder yields an orthogonal extension of existing approaches by providing a front-end to the modeler for defining the refactorings by modeling examples. The otherwise manually created refactoring descriptions are automatically generated from which representations in any language or formalism like graph transformation may be derived. Then it is possible to apply formal methods for analyzing the dependencies between refactorings as proposed by Mens.

Model transformation by-example. Defining model transformations rules by using the abstract syntax of graphical modeling languages comes on the one hand with the benefit of the generic applicability. On the other hand the creation of such transformation rules is often complicated and their readability is much lower compared to working with the concrete syntax as has been reported in several papers [3, 8, 19, 21]. As a solution, the usage of the concrete syntax for the definition of the transformation rules has be proposed like in ATOM^3 [8]. More recently, Baar and Whittle [3] discuss requirements and challenges how to define transformation rules in concrete syntax within current modeling environments. A specific approach of describing transformation rules for web application models is presented by Lechner [14]. In the field of aspect oriented modeling, transformations are also required for weaving aspect models into base models. Whittle et al. [23] present for example how to describe aspect composition specifications for UML models by using their concrete syntax. Summarizing, all these approaches significantly contribute to the field of the user-friendly development of transformations.

Strommer and Wimmer [19] as well as Varró [19] go one step further by defining transformations purely by-example, i.e., instead of developing transformation rules, an example input model and the corresponding output model are given. From these example pairs, the general transformation rules are derived by a reasoning component. Currently, the focus lies on model-to-model transformations between different languages, e.g., class diagrams to relational models. In-place transformations required for composite operations such as refactorings have not been considered by by-example approaches for models.

With the Operation Recorder we fill the gap between composite operation definition approaches and model transformation by-example approaches. Although the need for introducing refactorings by the user of modeling tools as well as the need for describing transformations in a more user-friendly way have been frequently reported, to the best to our knowledge, the Operation Recorder is the first attempt to tackle the by-example definition of model transformations representing composite operations such as refactorings. The only comparable work we are aware of is [18] which allows to define composite operations by-example for program code using the Squeak Smalltalk IDE. Although their general idea is similar to ours, three fundamental design differences exist, namely the Operation Recorder operates on models, is independent from any specific modeling language, and may be employed for any modeling environment.

7 Conclusions and future work

In this paper we introduced a tool for defining composite operations, such as refactorings, for software models in a user-friendly way. Modeling in the user's modeling language and in her environment of choice underlines this ease of use. Our by-example approach prevents modelers from acquiring deep knowledge about the metamodel and dedicated model transformation languages. The results of our evaluation emphasize the usability of our Operation Recorder because of minimizing the user's effort when defining such complex operations. The integration and usage of our Operation Recorder as component in a model versioning system leads to a reduction of conflicts and enables an intelligent conflict resolution.

In future work, we plan to enable the reuse of already defined refactorings for composing more complex refactorings. Preconditions of refactorings sometimes may include negative application conditions. For this reason, we will integrate the possibility of modeling forbidden model elements in order to match for non-existence of elements within preconditions. In a further step, translating operation specification models to graph transformations allows the critical pairs analysis and, thus, the detection of conflicts between refactorings. Finally, we would like to extend the operation specification model by adding smells—indicating problematic model fragments [12]—that may be solved using the defined refactorings.

References

1. AMOR Project Website, May 2009. `http://www.modelversioning.org`.
2. K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR - Towards Adaptable Model Versioning. In *1st Int. Workshop on Model Co-Evolution and Consistency Management, MCCM'08 @ MoDELS'08*, 2008.
3. T. Baar and J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *Ershov Memorial Conf.*, volume 4378 of *LNCS*. Springer, 2007.
4. E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *9th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'06*, volume 4199 of *LNCS*. Springer, 2006.
5. M. Boger, T. Sturm, and P. Fragemann. Refactoring Browser for UML. In *Int. Conf. NetObjectDays, NODe'02*, volume 2591 of *LNCS*. Springer, 2002.
6. P. Brosch, P. Langer, M. Seidl, and M. Wimmer. Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In *Int. Workshop on Comparison and Versioning of Software Models, MCVS'09 @ ICSE'09*. IEEE, 2009.
7. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley, 2003.
8. J. de Lara and H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-modelling. In *5th Int. Conf. on Fundamental Approaches to Software Engineering, FASE'02*, volume 2306 of *LNCS*. Springer, 2002.
9. D. Dig, T. N. Nguyen, K. Manzoor, and R. Johnson. MolhadoRef: A Refactoring-aware Software Configuration Management Tool. In *21st Conf. on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'06*. ACM, 2006.
10. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, 1995.
12. P. V. Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent uml refactorings. In *6th Int. Conf on the Unified Modeling Language, UML'03*, volume 2863 of *LNCS*. Springer, 2003.

13. D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.
14. S. Lechner. *Web-scheme Transformers By-Example.* PhD thesis, Johannes Kepler University Linz, 2004.
15. T. Mens. On the use of graph transformations for model refactoring. In *Int. Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE'05*, volume 4143 of *LNCS*. Springer, 2006.
16. T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.
17. I. Porres. Rule-based update transformations and their application to model refactorings. *Software and System Modeling*, 4(4):368–385, 2005.
18. R. Robbes and M. Lanza. Example-Based Program Transformation. In *11th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'08*, volume 5301 of *LNCS*. Springer, 2008.
19. M. Strommer and M. Wimmer. A Framework for Model Transformation By-Example: Concepts and Tool Support. In *Objects, Components, Models and Patterns, TOOLS'08*, volume 11 of *LNBIP*. Springer, 2008.
20. G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML Models. In *4th Int. Conf. on the Unified Modeling Language, UML'01*, volume 2185 of *LNCS*. Springer, 2001.
21. D. Varró. Model Transformation by Example. In *9th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'06*, volume 4199 of *LNCS*. Springer, 2006.
22. M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A Scripting Language for Refactoring. In *28th Int. Conf. on Software Engineering, ICSE'06*. ACM, 2006.
23. J. Whittle, A. Moreira, J. Araújo, P. K. Jayaraman, A. M. Elkhodary, and R. Rabbi. An Expressive Aspect Composition Language for UML State Diagrams. In *10th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'07*, volume 4735 of *LNCS*. Springer, 2007.
24. J. Zhang, Y. Lin, and J. Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In *Model-driven Software Development—Research and Practice in Software Engineering*. Springer, 2005.