Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder*

Petra Brosch†, Philip Langer, Martina Seidl, and Manuel Wimmer

Institute of Software Technology and Interactive Systems
Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at

Abstract

For the realization of language-independent, effective, and user-friendly model versioning systems, generic and efficient conflict detection is essential for correct and complete identification of conflicts caused by parallel modifications on one artifact. Usually, the genericity of the conflict detection costs a high price: language-specific operations and refactorings often remain undetected. Consequently, conflicts are not found or conflicts are wrongly indicated. To improve the quality of conflict detection, language-specific features have to be added. This involves usually much programming effort. We present a descriptive approach to define language-specific operations and refactorings by macro recording which allows an easy integration in generic conflict detection components.

1. Introduction

In this paper we present the By-Example Operation Recorder, a by-example approach for defining language-specific composite operations and refactorings. This approach is particularly tailored to allow end-user adaptation of generic model versioning frameworks to increase effectiveness in conflict detection as proposed for AMOR [1].

With the rise of model-driven software development in the last years, models are considered as first class entities in the software development process. Models are no longer just for documentation—they are the basis for generating executable software. Therefore, a suitable version control system (VCS) for models is crucial for an effective collaborative software development process. Because of the graph-based nature of models, conventional line oriented VCSs are not suitable for model versioning. Especially, conflict detection and conflict resolution must be adapted to fit for models.

In order to resolve not only textual and syntactic conflicts, but also semantic conflicts [3] composite operations must be detected. Therefore, we distinguish between *local and global conflicts* in model versions. Local conflicts may occur by concurrent changes of the same model element. Global conflicts occur when composite operations are performed as it is often the case during refactoring phases. Without the knowledge of composite operations, the merge cannot be performed because the current state of the model has changed too much. This may happen when elements no longer exist, or when the new model version violates integrity constraints [5].

Simple refactorings usually lead to small changes in the model which can also be handled by ID based VCSs. Such systems depend on unique IDs assigned to the model elements for the identification of differences and similarities between the original model and the working copies. Examples of simple refactorings are Extract Method or Pull Up Field[1]. However, doing refactoring is cleaning up the structure of the model by applying a series of refactorings to improve the quality of the overall software design [4]. Therefore, refactorings may include other refactorings, e.g. Extract Class is composed of Move Method and Move Field. For reliable conflict resolution of parallel global changes, knowledge about the composite operations of the refactoring is a must.

If refactorings are detected by the VCS, merge conflicts may be resolved automatically, or at least semi-automatically by supporting the user with more intelligent suggestions during manual resolution.

For instance, one modeler changes class `A` to singleton, i.e., only one instance of this class is created per runtime which is accessed by a public and static method named `getInstance()`. At the same time, a second modeler

*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.

[1] Refactoring Catalog: http://www.refactoring.com/catalog/index.html

performs some changes in another class and adds an instance of class `A` by calling the constructor. These changes are conflicting because in the merged version, the constructor is not visible to this class. If the changes of converting a class to singleton are detected as refactoring pattern, an appropriate conflict resolution pattern may be applied: accept both changes, but change calling the constructor to calling `getInstance()`.

Detecting refactorings may be realized by including refactoring patterns in the editor and tracking the user's operations. Operation-based conflict detection is very precise, because every change is recorded and the complete operation sequence may be replayed in the correct order. However, the main drawbacks are the loss of editor and metamodel independence and the limitation to the predefined refactoring patterns. To overcome theses drawbacks, a generic—with respect to Ecore-based metamodels[2]— and state-based approach with well-defined extension points for the definition of refactoring patterns on demand is highly valuable. This approach comes with the additional benefit of having a single source of information for pattern maintenance, when user-defined refactoring patterns are stored at the VCS server.

The rest of the paper is structured as follows. In Section 2, we continue with the motivation for applying a by-example approach. Subsequently, in Section 3 we give an overview on the underlying operation definition process of the By-Example Operation Recorder. Section 4 illustrates the approach with an example, followed by the technical description of its realization in Section 5. Related work is discussed in Section 6 and finally, in Section 7 we conclude and give an outlook on future work.

2. Motivating the By-Example Approach

On the one hand, a generic model versioning framework is demanded, because of its tool independence and applicability to arbitrary languages. This is especially needed when domain specific modeling languages are used. On the other hand, language-specific knowledge is required for improving the quality of conflict detection. Detecting composite operations such as refactorings needs a binding to the language-specific modeling concepts. Therefore, the generic framework has to be extended for each language with specific detection rules.

The By-Example Operation Recorder allows the language-specific adaptation of a generic model versioning framework by the end-user, i.e., the modeler. To simplify the modeler's life, adaptation should be possible by techniques close to modeling and only minimal programming effort should be required. With the By-Example Operation Recorder composite operations are defined by demonstration in combination with configuration. The by-example approach uses descriptive model artifacts shown in their concrete syntax which are annotated with some constraints to describe composite operations in model evolution.

Beside the usability aspect, there are two major requirements to the composite operation definition mechanism. First, it has to be complete, i.e., not to restrict the user in describing composite operations. Second, all atomic operations should be preserved and composite operations act as additional information to the generic conflict detection component only, in order to ensure the genericity of the approach.

The definition of composite operations in a by-example manner is the first building block for an adaptable operation-based model versioning system and the focus of this paper. However, we also want to employ by-example techniques in the future for configuring the conflict detection as well as the conflict resolution.

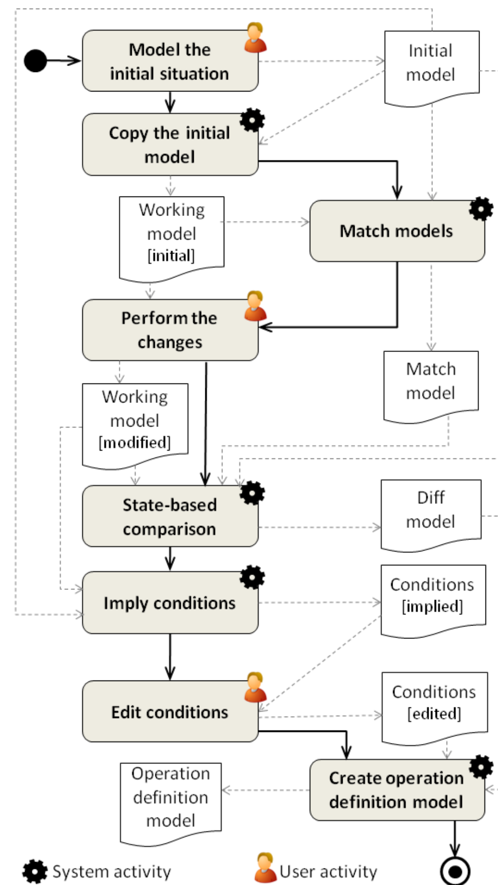

Figure 1. The Operation Definition Process

[2] http://www.eclipse.org/modeling/emf/

3. Operation Definition Process at a Glance

In this section we outline the 8-step operation definition process depicted in Figure 1 which is supported by the By-Example Operation Recorder. First, the user models the initial situation, i.e., the state of the model which is required to apply the composite operation. The output of this step is the *initial model*. In the next step, each element of the *initial model* is annotated with an ID and a *working model*, i.e., a copy of the *initial model*, is created. The IDs preserve the relationship of the original elements and changed elements in the working copy even after intense modification. Consequently, the generated *match model* between the initial model and working model is exact and complete. Then, the user performs the complete composite operation on the working model by applying the atomic operations step-by-step. Based on the previously generated match model, the atomic operations are determined automatically using a state-based comparison and the results are saved in the *diff model*. Subsequently, the *pre-* and *postconditions* are inferred by analyzing the initial model and working model, respectively. Usually, the automatically generated conditions 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 definition process and are refined by the user by configuration. In particular, parts of the conditions can be activated and deactivated within a dedicated environment with one mouse click. Finally, the *operation definition model* is generated which consists of information from the diff model and of the edited pre- and postconditions. This operation definition model is used further on for the detection of the defined composite operation during the versioning process.

4. By-Example Operation Recorder in Action

In this section we describe by-example how composite operations are defined with the help of the By-Example Operation Recorder. Step by step we explain how the operation “convert to singelton” is defined in the following. This operation changes a class in such a manner that only one instance of this class exists per runtime environment. Therefore, the constructor is set to private. Consequently, the constructor cannot be called by any other instance. In order to obtain an instance of this class, a static method `getInstance()` is added which returns the only existing instance of this class. In the following, it is assumed that `A` is the name of the class the refactoring is applied to.

The graphical user interface (GUI) which is shown in Figure 2 is the frontend of the By-Example Operation Recorder. This GUI supports the afore presented operation definition process of Section 3. In the following, we

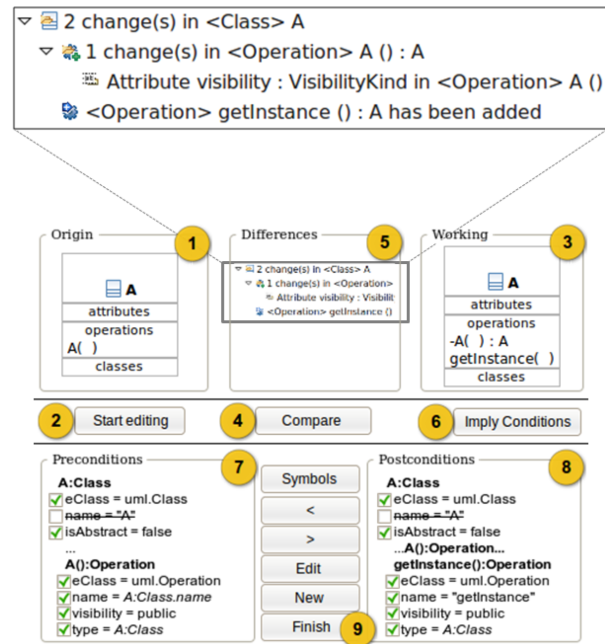

Figure 2. GUI of the By-Example Operation Recorder

describe the nine steps performed by the user in order to create a new composite operation.

1. *Model the initial situation.* In the *Origin area* of the GUI, all aspects relevant to execute the composite operation are modeled. In our example we create a class `A` with a public constructor `A()`. This is the model state before the refactoring is applied.

2. *Copy the initial model.* When all necessary aspects of Step 1 are modeled, the button Start Editing has to be pushed. This action copies the content of the *Origin area* into the *Working area* of the GUI. Naturally, the original model and its copy match 1:1, i.e., the match is complete and exact.

3. *Perform the changes.* Now the user performs the atomic changes of which the composite operation consists on the working copy. In our example the constructor is set to private and the new method `getInstance()` is added.

4. *Calculate the differences.* When all changes of the composite operation have been performed by the user, they are confirmed by pushing the button Compare. A generic, state-based comparison based on the perfect match generated in Step 2 is applied.

5. *Report of the performed operations.* Now the previously recognized atomic operations are presented in the *Differences area* of the GUI. For the ease of readability, the upper part of Figure 2 contains a zoom into this area; and indeed, as expected the two changes concerning the visibility of the constructor and the addition of the new method `getInstance()` have been detected.

6. *Imply the necessary conditions.* As all operations have been executed and identified, it is now possible to infer all necessary pre- and postconditions for the composite operation. For the definition of the constraints, a context element has to be chosen. In our example there is only one element, i.e., class `A`, as potential context element, so it is *automatically* selected. If there are multiple elements at the root level, the user must choose the context for the constraints. When the button Imply Conditions is pushed, potential constraints are created and shown in the *Preconditions area* and, respectively, in the *Postconditions area* of the GUI. There each model element is identified by a variable symbol. For example, the symbol `A:Class` represents the class `A`, and `A():Operation` represents the constructor. The purpose of these symbols is to generally characterize the role of a model element within the composite operation. For example, `getInstance():Operation` universally describes the method in the working model returning the single instance of `A`. The symbol names are automatically set using the naming convention `<Name>:<Type>`. For convenience, the user is able to rename generated symbols. The scope of a symbol is either the origin model or the working model. Nevertheless it is possible to access symbols of the opposite model in the conditions using the prefixes `origin:` and `working:`, respectively. The symbols are arranged in a tree to indicate their containment relationships. In the final two steps, the conditions are refined by the user as the automatically generated constraints will usually be too strong. For example, the refactoring is currently defined for classes named as `A`.

7. *Edit the preconditions.* In this step, the user leverages the implicitly defined constraints in such a manner that they represent the necessary preconditions for the execution of the composite operation. In our example three changes are necessary. First, the operation is applicable not only to class `A`, but to any arbitrary class. Therefore, the checkbox `A:Class.name = "A"` must be unchecked. Second, the constructor should not have the name of the example class `A`, but the name of the class `A:Class`. Consequently, the condition `A():Operation.name = "A"` has to be changed to `A():Operation.name = A:Class.name`. Third, the return type of `A():Operation` has also to be set to `A:Class` instead of `A`.

8. *Edit the postconditions.* This step is very similar to the previous step with the difference that not the preconditions but the postconditions must be manually processed. In our example four properties are affected. Again the occurrences of the reference to the example class `A` in the name attribute of `A:Class` and in the name and type attributes of `A():Operation`, respectively, are changed to the general symbol `A:Class`. Finally, the type of the return value of the newly introduced operation `getInstance():Operation` is also generalized to `A:Class`. The change of the visibility property of the constructor from public to private has been automatically inferred from the working copy.

9. *Create operation definition model.* In this last step the user confirms the changes of the pre- and postconditions by pushing the Finish button. Finally, the operation definition model is stored persistently.

5. Realization

The By-Example Operation Recorder presented in the previous section is realized based on the Eclipse Modeling Framework[3] (EMF) and EMF Compare[4]. Therefore, in the *Origin area* as well as in the *Working area* of the GUI any EMF-based editor might be applied for the creation of the example and for the description of the refactoring operation. This allows the user to define the model in his/her favorite modeling environment.

After editing the working copy, a two-way state-based comparison is performed by EMF Compare (the button Compare is pushed). For the complete and correct determination of the necessary atomic operations, we abandon the heuristic methods of EMF Compare. Instead, the 1:1 match obtained after Step 2 is kept in memory during the whole operation definition process. This allows a precise mapping of the elements of the left-hand side and right-hand side models even after significant changes.

In Step 6, variable symbols and constraints are inferred from the models. For their representation we use the Object Constraint Language (OCL), which is a very popular language for defining constraints on models. For the definition of variable symbols we apply OCL `def`. An example is shown in Figure 3.

As already mentioned, the selection of a context element is necessary to which the OCL expressions refer. In the ex-

[3]http://www.eclipse.org/modeling/emf/
[4]http://www.eclipse.org/modeling/emft/

```
/* Symbol definitions */
1 def: elm1      : Class     = self
2 def: elm1_sub1 : Operation =
         self.ownedOperation()->at(1)
3 def: elm1_sub2 : Operation =
         self.ownedOperation()->at(2)

/* Postconditions */
4 elm1.isAbstract = false
5 elm1_sub2.name  = 'getInstance'
6 ...
```

Figure 3. Constraints and symbols in OCL

ample of Figure 3 `self` represents the class `A` in the working model and is assigned the symbolic name `elm1`. In the second line, the first model element contained by `A` is named `elm1_sub1`. The symbol names in OCL are only used internally. To increase the usability they are mapped to more user friendly names which are displayed in the user interface. For instance, the internal name `elm1` is shown as `A:Class` in the user interface illustrated in Figure 2. When all model elements have been assigned to symbols, the automated generation and user-defined modification of the constraints is possible. Line 5 illustrates the condition fixing the name `getInstance` of the operation returning the only instance of `A`.

In future, we plan to add a possibility to include user-defined definitions (button New in Figure 2). For the definition of refactorings whose number of atomic operations is not constant, mechanisms will have to be added which allow the user to specify repetitions. For example, it must be possible to apply an operation on all methods of a class. Furthermore, in the current approach it is not possible to provide alternative descriptions of one refactoring. To overcome this shortage, boolean connectives like *logical or* and *logical and* have to be provided for combining and excluding the atomic operations, respectively.

Overall, the By-Example Operation Recorder allows the description of composite operations in terms of a difference pattern, preconditions, and postconditions. On this basis, we are now able to detect such an operation within a generic difference model resulting from the state-based comparison of two models. First, difference patterns have to be identified. If a difference pattern is recognized within the difference model, then pre- and postconditions must be checked. If all constraints hold, then it can be assumed that the composite operation has been applied and the difference model is annotated with the information that for example a refactoring has been applied.

6. Related Work

By-example approaches have a long history in computer science and can be traced back to Zloofs Query-By-Example (QBE) approach from 1975 [11] for querying relational databases. A more related by-example approach comes from Lechner [6] by extending Zloofs QBE approach in order to enable the definition of transformations for web application models. One of the novelties of his work is that Lechner introduced a generation part using template definitions in addition to the query part which is closely related to our approach. However, there are also two major differences. First, in contrast to Lechner, we do not introduce additional notation elements for defining the templates in a generic way. Instead, we only use the notation elements of the modeling language and separate variables from constants in additional tree viewers. This allows us to provide a completely generic approach and tool support which is not bound to a modeling language, domain, or environment. Second, the application context of the two by-example approaches is completely disjoint. While Lechner focus on web application patterns only, we define more generic patterns such as refactoring or design patterns which are typically applied on UML-like models.

Refactorings are an integral part of software development (for an overview see [8]). Usually refactoring operations are hard coded for specific programming languages and so they cannot be reused in any other context than the intended one.

Verbaere et al. defined in [10] the domain-specific, functional refactoring language JunGL which operates on the graph representation of a program. In contrast to our approach, the intention behind JunGL is not the detection of already performed refactoring operations, but the application of refactorings on given programs. The By-Example Operation Recorder could be a user-friendly way to define JunGL expressions.

Mens et al. [7] present an approach to define refactorings by typed attribute graph transformations. Then they apply critical pair analysis in order to detect conflicts between refactorings performed by different developers. Graph transformation rules can be seen as a way to define model transformations by-example. In particular, a graph transformation rule consists of a precondition and a postcondition describing a generic transformation example. However, the generic examples are described in the abstract syntax of the modeling languages which requires in-depth knowledge of the metamodel. Most modelers only have knowledge of the modeling notation but not of the internal structures used for automatic processing of the models. Therefore, in contrast to graph transformation, we focus on the definition of composite operations by the end-user which is only aware of the modeling notation.

Recently, several dedicated model versioning systems like COMOVER [2] and Odyssey-VCS [9] have been proposed, but to the best of our knowledge currently there exists no system which is extensible with respect to composite operation detection.

7. Conclusion

In this paper we presented an example-driven approach for the user-friendly definition of composite, language-specific operations. The detection of such operations creates the prerequisites for effective model comparison which constitutes one of the core components of model versioning systems. The aggregation of atomic and abstract operations to complex and composite operations such as refactorings allows a considerable simplification of the underlying difference model. The additional, language-specific information prevents the detection of wrongly indicated conflicts between compared (versions of) models or it allows at least more sophisticated resolution support. Furthermore, the integration of new operations in a generic model comparison and conflict detection framework is achieved in a completely declarative manner (i.e., no programming effort is necessary).

In future work, we plan to incorporate the by-example operation definition as an integral component in the model versioning framework AMOR [1]. We expect the example-driven method to shape up as a powerful tool to realize one of the most urgent requirements on AMOR's architecture: the adaptability to any modeling language.

References

[1] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR—Towards Adaptable Model Versioning. In *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management*, 2008.

[2] I. Barone, A. D. Lucia, F. Fasano, E. Rullo, G. Scanniello, and G. Tortora. COMOVER: Concurrent model versioning. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 462–463, 2008.

[3] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Compututing Surveys*, 30(2):232–282, 1998.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] M. Kögel. Towards Software Configuration Management for Unified Models. In *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*, pages 19–24. ACM, 2008.

[6] S. Lechner. *Web-scheme Transformers By-Example*. PhD thesis, Johannes Kepler University Linz, 2004.

[7] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. In *Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions*, 2005.

[8] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[9] H. Oliveira, L. Murta, and C. Werner. Odyssey-VCS: a Flexible Version Control System for UML Model Elements. In *Proceedings of the 12th International Workshop on Software Configuration Management*, pages 1–16. ACM, 2005.

[10] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a Scripting Language for Refactoring. In *Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, 2006.

[11] M. M. Zloof. Query by example. In *Proceedings of the National Compute Conference*, pages 431–438, 1975.