Determining the Intent of Code Changes to Sustain Attached Model Information During Code Evolution

Michael Langhammer, Max E. Kramer
Karlsruhe Institute of Technology
{michael.langhammer, max.e.kramer}@kit.edu

Abstract

If code is linked to models with additional information that cannot be represented in the code, changes in the code may have unwanted effects on these models. In such scenarios, the desired effect of code changes may be unclear and impossible to determine regardless of the used change recording or propagation mechanism. Existing round-trip engineering tools do not solve this problem as they just support models that contain only information that can be regenerated from the code or drop such information. In this position paper, we propose an approach for controlled code evolution that automatically maps unambiguous code changes to well-defined operations and that blocks ambiguous changes until the user clarified his intent. We present a case study for Java code and component-based architectures to show how such an approach would only permit code changes with unambiguous effects on the architecture. To stimulate discussions at the workshop, we argue why such an approach is necessary and describe benefits and drawbacks of such a solution.

1 Introduction

In Model-Driven Software Development (MDSD), a software system may be described by its source code and with models that represent a part of the system from a specific perspective for a specific task. The used modelling languages and model instances may be tailored to specific development and analysis tasks so that they display only the required information. If the same information is required for several tasks, then information may be spread across various models. In such cases, all artefacts have to be mutually synchronized to avoid differences between code and models (drift) and inconsistencies (erosion) during software evolution. There are several ways to bypass this need for synchronization: Redundant information can be completely disallowed or strict refinement can be employed so that every information has a unique origin and redundant elements in other artefacts can always be regenerated [3]. A problem arises, however, if attached models contain information that can neither be represented nor be computed from the code. In such cases, code changes may occur for which the intended impact on attached models is unclear.

2 Approach

We propose a co-evolution approach that differentiates between unambiguous and ambiguous code changes with respect to a linked model. We use the Java Model Printer and Parser (JaMoPP) [4] to obtain a syntax tree model so that we can directly work on code model changes instead of textual changes.

Unambiguous changes are changes where the impact to the linked model and the intent of the user are clear. For example, when the effect on the linked model has been specified upfront in a transformation. This means a change is unambiguous when the corresponding operations on the linked models are clear.

Ambiguous changes, however, are changes where the intent of the developer cannot be determined automatically. In our approach, these changes are blocked to let the developer clarify his intent in a way that unambiguously induces an operation on the attached models. If a developer moves, for example, a method from one class to another, additional information on design rationale, which is not represented in the code but in attached models, may become obsolete for the new system. As the developer has no access to this information in the code editor, it is unclear which effect on this information he desired for this code change. The desired effect of his change cannot be determined with confidence without any further information or assumptions regardless of the used change recording or propagation mechanism. For our method move example, this means, that the developer can specify whether the former design rationale for the method is still valid. After this, the source code change is mapped to an operation on the syntax model and can be propagated to the linked models using model-to-model transformations that use the additional clarification information.

If a developer cannot or does not want to clarify the impact on linked models, we may obtain inconsistencies that have to be resolved later or by others.

The benefit of our approach is that models, which contain additional information with unclear code-correspondence, are kept synchronized with the source code. Hence the effort for manually synchronization between the model and source code is reduced to the provision of clarifying information. Since we do not

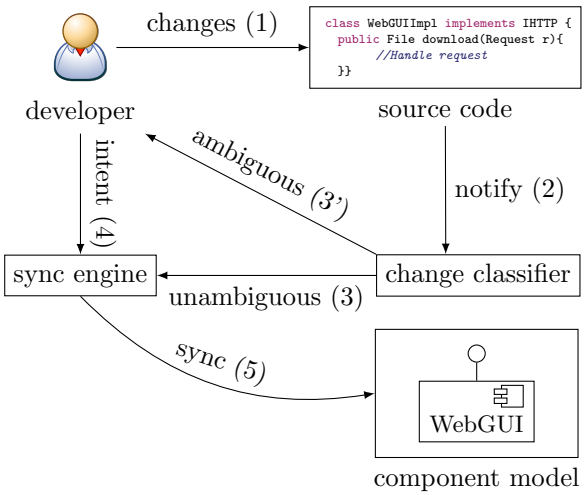

Figure 1: The proposed approach applied to Java source and a component-based architecture model.

regenerate the linked model, additional information that is not represented in the source code is sustained in the models. Furthermore, developers are notified when their changes affect linked models, which gives developers the possibility to influence the impact on linked models. A drawback of our approach is that the workflow of developers may get interrupted when the changes they made are blocked until clarification.

We will embed the proposed approach into a view-centric engineering framework [5]. Change implications will be specified with declarative metamodel mappings and instance level correspondences between code regions and linked models will be automatically managed in a tracing model. Using this information we will check whether the current code change affects code elements that are linked to another model.

3 Application scenario

An application scenario of the proposed approach is to synchronize Java source code and component-based architecture models (see Figure 1) with additional performance information from the Palladio simulator [2]. Palladio models consist of components, interfaces and signatures, which are represented in the source code e.g. in terms of interfaces, classes and method declarations (cf. [5]). If a developer changes (1), for example, the name of an architecture relevant interface, then this change is unambiguous (2,3) and the corresponding interface in the component model can be renamed automatically (5). If a developer renames (1), for example, a class method that implements a signature of an architecture relevant interface, then the impact on the component architecture is not clear (2). The developer has to be asked (3') whether he wants to change the signature of the component interface or whether the changed method should no longer be linked to the architecture interface (4). If he chooses the first option, this will also influence other parts of the code: all corresponding methods of classes that realize components that provide the corresponding architecture interface will be renamed as well (5).

4 Related Work

The Software Model Extractor (SoMoX) [3] is able to generate component models from source code. The architecture is, however, not updated incrementally and additional information added manually to the generated architectureis lost during regeneration. UML-Lab[1] supports round-trip engineering for UML class diagrams and source code. It does, however, not support additional information in the class diagrams which can not be represented in the source code. The Orthographic Software Modeling (OSM) [1] approach proposes the use of a Single Underlying Model (SUM) that contains all information of a particular software system. This approach does not need to synchronize information between models because there is no redundant information in the SUM. It is, however, an open question how a SUM for object-oriented code and component-based architectures can be obtained.

5 Conclusion

In this position paper, we have presented an approach for the co-evolution of source code and models that contain additional information. We have also briefly described an application scenario using Java source code and component-based architecture models. In the future, the presented approach could be the foundation for a round-trip engineering approach in which code and models can evolve in parallel even if they partly contain independent information.

References

[1] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering*. Vol. 69. 2010, pp. 206–219.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[3] Steffen Becker et al. "Reverse engineering component models for quality predictions". In: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE. 2010.

[4] Florian Heidenreich et al. "Closing the gap between modelling and java". In: *Software Language Engineering*. Springer, 2010, pp. 374–383.

[5] Max E Kramer, Erik Burger, and Michael Langhammer. "View-centric engineering with synchronized heterogeneous models". In: *Proceedings of the 1st Workshop on VAO*. ACM. 2013.

[1] http://www.uml-lab.com/