Towards a Model-Based Software Reengineering Approach with Explicit Behavior Descriptions: Chances and Challenges

Marco Konersmann, Bernhard Rumpe
Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

Introduction Model-based software reengineering (SRE) uses a horse-shoe process style to modernize original systems [4]. A model of the original system is constructed, adapted, and the target system code is partly or as a whole generated. These translations can introduce faults and quality issues. When the model is incorrect with respect to the original system, the target system might have missing functionality or bad quality. We can compare the target system to the original system, but this does not show error sources or how to fix them: is the issue in the model extraction, the model adaptation, or the code generation?

A major kind of models used in model-based SRE are software architecture (SA) models. SA is often expressed as informal boxes and lines, alongside textual documentation for communication. While this provides a large flexibility for the modelers to communicate, these models cannot be processed automatically for analysis or construction of the system. The Unified Modeling Language (UML) is broadly used for SA and design. It provides a rich syntax for modeling architectures and some degree of formality for automated processing. However, the mapping of UML elements to an implementation or a runtime behavior is not clearly specified. E.g., there are many potential implementations of UML components or their interconnection, depending on the technical domain and intended level of abstraction.

Several approaches exist for creating suitable architecture models of a system, e.g., using static or dynamic analysis [2]. However, since there is no universally accepted SA language [6], reuse is limited to related projects. Using a modular SA language with an explicit behavior description, that is suitable for the diverse concerns of SA, can help increasing reuse. In this paper we sketch a reengineering process utilizing a model-based approach with explicit behavior descriptions, and discuss the chances and challenges.

Horse-Shoe Model using Software Architecture Models with Behavior Descriptions Figure 1 shows a model-based SRE process model that allows for simulation due to SA models with explicit behavior descriptions. The original system comprises the code, run time data such as logs and performance annotations, and further artifacts, such as design decision documents and their rationale. First, we create an architectural model of the original system alongside trace links from the code to the model elements. The approach requires an architecture modeling language that defines behavior of components and their interconnection, and the data processing within the components, in a processable way. The architecture model is used to simulate the system and the quality properties are tuned until the simulation results resemble the original run time data sufficiently. It can then be adapted to resemble the intended target architecture, and simulated to evaluate the run time quality compared to the architecture model of the original system. Then, code is generated from the target architecture model. The process maintains trace links in each step. Therefore the relation of original system artifacts to the target system can be traced. The SRE process might introduce the need to update or refactor these artifacts, e.g., when non-architectural code needs to be adapted or design decisions need to be revised.

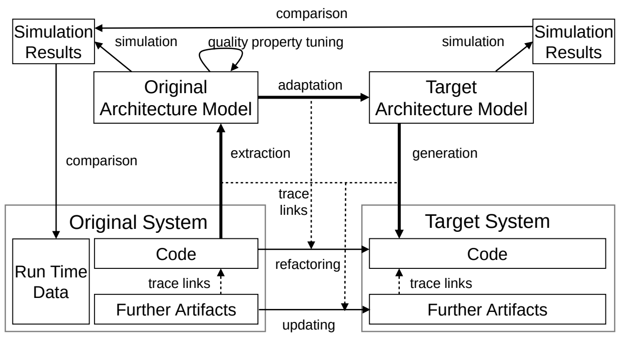

Figure 1: Horse shoe model for software reengineering with model-based software architecture

Chances We see the following chances in applying the approach: First, using a model-based approach allows to automate the SRE process: This includes to automate the creation of architecture models from code, model transformations, and code generation. We require languages that provide a clear interpretation of their behavior description, that holds beyond individual projects or even organizations. They therefore allow to define reusable transformations, e.g., to replace function calls with event-based communication or to split or merge components. These reusable transformations can be provided alongside the architecture language and reused in multiple SRE projects.

Second, an explicit behavior description enables the simulation of the modeled architecture, for comparing it against the original system. This can increase the quality of the target architecture model simulation.

Third, an architecture modeling language with explicit behavior descriptions allows to analyze the data and control flow through the modeled system. This can help understanding the original system and how to improve the architecture for a target system.

Finally, as the process builds trace links between the code and the architecture model, trace links of existing other artifacts to the original code can be translated into trace links between these artifacts and the target code. These can be used to reference artifacts that need to be updated in the SRE process.

Challenges There are some challenges regarding the approach, that we need to address: First, architectures today have to cope with a large diversity, e.g., in styles, platforms, or implementation languages, which is one of the reasons why no SA language exists, that is suitable for all projects [6]. The diversity also imposes a large complexity on the automated model extraction. Hence, an implementation of this approach needs to carefully choose the architecture language. An implementation should take into account that architectural concepts and technologies evolve. We plan to define fine-grained requirements for architectural languages, model-extraction and code-generation mechanisms to enable efficient reuse or adaption of existing languages and tools.

Second, the simulation of architectures requires a suitable level of abstraction. Finding the right level of abstraction depends on the purpose of the modeling, and thus on the purpose of the reengineering project. E.g., if the purpose is only to wrap the original system in new interfaces, then the level of abstraction might be higher, than if an original system should be reimplemented in a different paradigm. This requires the architecture language, the model extraction mechanism, and the code generator to handle different levels of abstraction, e.g., projects in a repository, classes, or deployments, within the same reengineering project.

Third, creating an architecture model with the quality properties of the original system requires explicit behavior descriptions and information about the modeled system. Among others, this depends on the quality properties to validate, the platform, and the code. One possibility to tune the quality parameters of the model is the use of reinforcement learning.

Fourth, creating traceability is a challenge in all but trivial projects. In practice, trace links require considerable effort to maintain [1]. Automated model extraction and code generation can automatically create trace links. Automation can use naming conventions or other external knowledge to improve the degree of automation for further artifacts.

Finally, refactoring existing, non-architectural code and updating other artifacts automatically is a challenge. As code is formal, elements can be referenced and translated using automated processing. Other artifacts—e.g., design decisions documents—are less formal. Hence there is a greater challenge for automated updates. The presented approach can point to documents or parts therein to reconsider by following the trace links, to decrease the manual effort.

To evaluate the functional suitability, we plan to apply a demonstrator to information system use cases. While we consider the approach domain-agnostic, the extraction mechanisms and code generator need to cover domain-specific frameworks and coding styles. We propose to use MontiArc [3] as a SA language for this approach, because it has explicit behavior descriptions, allows to model different levels of abstractions, an can easily be extended with domain-specific refinements. We can then provide libraries and extensions to represent common technologies and concepts. Codeling [5] is a tool for architecture model extraction and code generation, which considers diversity in the technologies and concepts of architectures. Codeling automatically creates trace links when extracting architecture models. Further trace links are considered an input to this approach, meaning that better trace links can lead to better results. We plan to adapt Codeling to handle MontiArc models and to handle diversity within single projects.

Conclusion We presented an approach for software reengineering using architecture models with explicit behavior descriptions, and discussed its chances and challenges. Future work can include specifying a reference architecture for the approach and providing an implementation to showcase the functional suitability.

References

[1] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, June 2007.

[2] Stephane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[3] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.* Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[4] R. Kazman, S.G. Woods, and S.J. Carriere. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, 1998.

[5] Marco Konersmann. *Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code.* phdthesis, University of Duisburg-Essen, March 2018.

[6] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.