Automated Reverse Engineering of the Technology-Induced Software System Structure

Yves R. Kirschner, Jan Keim, Nico Peter, and Anne Koziolek

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{yves.kirschner, jan.keim, anne.koziolek}@kit.edu,
nico.peter@student.kit.edu

Abstract. Evolving system architectures can be complex and difficult to understand, leading to problems such as poor maintainability. Automated reverse engineering of system structure models from source code can mitigate these problems and facilitate architectural decisions. However, identifying components with their interfaces can be challenging because components are often implemented in different frameworks and interconnected in complex ways. Our approach aims to create software models using reusable concept descriptions for reverse engineering. We use structural-level mapping rules to reconstruct the static system structure from source code, assuming that the technology used can determine the components with their interfaces and deployment. We evaluate our approach on four public reference systems. The analyses show that technology-specific rules already provide good results, but the addition of project-specific rules further improves reverse engineering.

Keywords: Automated reverse engineering · Reusable concept descriptions · Software architecture models · Structure-level mapping rules.

1 Introduction

Large software systems often become complex and difficult to understand during development, leading to problems such as poor maintainability. To solve these problems, software architects must have a clear understanding of the system's software architecture. Therefore, there is a growing need for automated tools that can extract the static system structure from source code. Automated reverse engineering of models offers benefits such as improved software maintainability and support for architectural decisions. However, reverse engineering the static system structure is challenging due to difficulties in identifying components and their interfaces [4]. Factors such as different programming languages and frameworks used to implement components make it difficult to accurately capture all possibilities. In addition, components may have complex interconnections, making it unclear which interfaces they use to communicate. The goal of our proposed approach is to support the creation and maintenance of models. To reverse engineer models, we want to utilize reusable concept descriptions. To achieve this goal, we formulate the following research questions: *RQ1:* To what

extent do a selected framework impact the underlying software static system structure? *RQ2:* How can this impact be implemented as a transformation for reverse engineering? Our new approach aims to use mapping rules at the structural level to reconstruct the static aspects of the system from source code. We assume that components with their interfaces and deployment can often be explicitly determined by the used technology, e. g., application frameworks. Since we want to use knowledge about technologies to discover components, we expect our approach to generate models that are more consistent with the present architecture. We also expect technology-specific rules to provide a better understanding of the relationships between a technology and its underlying concept and the static system structure.

2 Foundation

Web services are a type of software component that provide a standardized way for communication over the internet [13]. RESTful web services make resources available through a uniform and predefined set of stateless operations, typically HTTP methods [10]. *Reverse engineering* aims to identify structures in the form of elements and relationships within the software system under investigation. This involves analyzing and understanding software systems that may have evolved over time or may not be formally documented [9]. In this way, patterns, relationships, and constraints in the system can be identified, providing insight into the design, implementation, and maintenance of the software. Manual reverse engineering a software architecture can be difficult and time-consuming process, especially for large and complex systems. Automating this process typically involves analyzing the system's code to identify components, interfaces, dependencies, and other architecture elements. This can be accomplished through various techniques, e. g., static analyses, dynamic analyses, or both [5]. Static analyses examine the source code or compiled binary of the software system without running the system. They are fast but may miss certain runtime-related aspects of the system's behavior. Model-Driven Reverse Engineering (MDRE) is the task that focuses on recovering models. Favre et al. define MDRE as the creation of descriptive models from existing systems that have been created in some way [6]. MDRE is about transforming heterogeneous software development artifacts into homogeneous models.

3 Approach

Our goal is to automatically extract from existing software development artifacts the structure that represents the static aspects of the system. This structure includes the way a software system is split up into components and shows the dependencies among these components as well as the deployment of these components to nodes. We consider artifacts written during the development of a software system, e. g., source code or other configuration files like deployment descriptors. The idea is to model the knowledge of used technologies in order to

reverse engineer this static system structure from artifacts. This approach is an implementation-based reconstruction of the source code based on implementation knowledge and grouping based on deployment. Rules capture how a certain concept is implemented in a technology and how this concept affects the static structure of a system. These rules are expressed as model-to-model transformations. The technology-specific rules are developed by analyzing the patterns of each technology and mapping them to model elements. Rules can be formulated to cover any aspect of a technology that can be identified through static analysis of a software project's artifacts. Although the approach is designed for reuse, it also supports project-specific rules. These rules can be used, for example, to model how components are implemented in a specific software project. To define transformation rules, we use the transformation language Xtend [1]. Listing 1 depicts a simplified form of a extended Backus-Naur form (EBNF) that defines the framework of our approach for such rules. However, the shown EBNF is a simplified version and contains only a subset of the possible solutions. We only include the most relevant production rules.

```
<RulesDocument> ::= {<Rule>}
<Rule> ::= [<Loop>] {<Condition>} {<Detection>}
<Loop> ::= "for" <HelperGet> ":"
<Condition> ::= "if" <HelperBool> ":"
<HelperGet> ::= "getClasses()" | "getMethods()" | ...
<HelperBool> ::= "isAnnotatedWith()" | "isExtending()" | ...
<Detection> ::= "detectComponent()"  | "detectRole()" | ...
```

Listing 1: Simplified EBNF as a conceptual framework for our rules. The example represents non-terminals for helper methods with two methods each.

A `RulesDocument` is a collection of rules and a `Rule` consists of several non-terminals. A rule can start with a loop that iterates over parts of the code model instance. The objects to be iterated are defined in the non-terminal `HelperGet`. The next part of a rule is the `Condition` that is defined as an if-expression with a non-terminal `HelperBool` to select what to inspect. The rule engine provides predefined aspects of what a user might be looking for, such as specific annotations or names. The idea of these `HelperGet` and `-Bool` methods is to discover elements. These methods define queries to the code models that return elements of interest. These query methods need to be defined only once and can be reused without knowing the exact structure of the code models. To associate a model element with the current object, the user performs a detection that can identify, e. g., components, interfaces, and provided or required roles. The methods for these identifications are provided by the rule engine.

The first step in our reverse engineering approach is to create a model from existing artifacts that provides a unified view of the software system. In order for these models to provide a unified view, they must conform to a suitable given metamodel. The structure of these models is realized in MDRE by so-called

discoverers that depend on the associated metamodel. The second step is the main step of our reverse engineering approach. The previously created models are used to effectively achieve the desired reverse engineering scenario. During this step, these models are analyzed using rules represented by model-to-model transformations. For this purpose, our approach includes a framework for MDRE that allows for reusable acquisition of this knowledge. These two steps use the proposed rules to identify and compose relevant model elements and their relationships from the collection of code model elements. The first sub-step is for the inner component model and uses the results from model discovery. The defined rules allow the identification of software system elements, e. g., components, interfaces, and communication paths. The second sub-step is for the outer service model. Here, rules for the deployment descriptor artifacts are used to compose components into larger logical units. These composite components can represent services or subsystems.

To generate the static system structure model instance, the first model understanding sub-step is to create the interfaces by extracting information from the code model instances. To create a model, there need to be several individual rules, each of which helps to identify at least one model element. The previously defined rules are used to identify relevant elements within specific code structures. The rules formulate these structural patterns that define which code artifacts map to relevant components and how to infer provided and required roles. Extracting model information from the code model involves identifying the interface name, signature name, return type, and parameters of each method. The rules define which code artifacts are associated with relevant components and how to derive provided and required roles.

The second sub-step is to compose components based on the information in the deployment descriptor models into services. By associating each component with a service, subsystem boundaries can be inferred, resulting in a more readable and understandable view of the system. Delegation connectors link provided and required interfaces with inner components, while assembly connectors link inner components with each other. Assembly connectors are created for each component that matches the required role of another component in the service.

In the third step, all the information determined by the rules of the previous model understanding is merged into the final static system structure model. For our approach, we use the Palladio Component Model (PCM) [12] as a component-based architecture model, but the concepts are also applicable to others Architectural description languages (ADLs). Our approach can be adapted in this model generation step to use other notations for describing component-based architecture models, e. g., UML component or deployment diagrams.

4 Evaluation

For the evaluation, we first create our own reference model for the software system under study in order to compare it with our automated results. To do this, we first analyze the source code and configuration files to identify the compo-

nents and their roles. We then compare the constructed model with existing documentation or diagrams available in the repository or linked to the software system, and perform a refinement and validation of the constructed model using expert knowledge and domain-specific information. The extraction process involves applying the technology-specific rule set to the system and comparing the resulting model to the expected one. Success is measured using precision (p), recall (r), and $F1$, the harmonic mean of both. True positives are relevant elements that should be found, false positives are falsely extracted elements, and false negatives are missing elements in the extracted model relative to the expected elements. Recall alone does not guarantee correctness as misrecognized elements may still be present. Precision is also necessary to indicate misrecognition. The extracted software architecture is analyzed for possible improvement by adding new rules to the technology-specific rule set. The completeness of the generated model and the effort required to define new rules is measured by the number of newly defined rules and the total lines of code (LOC) required.

Spring Systems: The first two case studies are two open source systems implemented primarily based on the Spring framework. PetClinic[1] is a Spring Boot microservices application implemented with Java and Docker that simulates a simple management system for veterinary clinics. Piggy Metrics[2] is a simple financial advisory application developed to demonstrate the microservice architecture model using Spring Boot and Docker. The Spring PetClinic reference system consists of 4 microservices. For these combined, 11 components, 11 interfaces, 11 provider roles, and 9 required roles are evaluated, for a total of 42 elements. The model extracted by our approach with only the technology-specific rules contains all intended components, interfaces, and provided roles. However, 2 required roles are missing, 5 data types are incorrect in the correctly identified roles, and 3 data types are missing in the identified roles. The good results are due to the fact that the system adheres closely to the Spring specifications, providing a solid foundation for defining the technology-specific rule set. The Piggy Metrics reference system consists of four microservices. For these combined, 29 components, 28 interfaces, 28 provider roles, and 22 required roles are evaluated, for a total of 107 elements. The model extracted by our approach with only the technology-specific rules has a total of 116 correctly identified elements, 25 missing elements, and 12 incorrect elements. 23 components are correctly identified, while 5 interfaces and 6 provided roles are missing. In addition, 4 interfaces and 6 provided roles are incorrectly identified. Four required roles are also missing. New project-specific rules are derived to improve the extraction result by analyzing missing and incorrect elements. Two new rules are added, modifying the existing rules with a total of 7 new LOC. These new rules correctly identify the four missing components and their associated roles.

[1] https://github.com/spring-petclinic/spring-petclinic-microservices
[2] https://github.com/sqshq/PiggyMetrics

JAX-RS Systems: Tea Store[3] is a microservice application implemented in Java and Docker that emulates a simple web store for tea. Acme Air[4] is an application implemented in Java for a fictional airline. The Tea Store reference system consists of six microservices. For these combined, 79 components, 64 interfaces, 79 provider roles, and 10 required roles are evaluated, for a total of 232 elements. The model extracted with only the technology-specific rules has a total of 104 correctly identified elements, 128 missing elements, and no incorrect elements. The system was analyzed to identify missing elements, and new project-specific rules were applied to improve the extraction. In total, four new rules were created and implemented in 29 LOC. Using these new project-specific rules, the new model extracted by our approach produced 15 false positives, 13 false negatives, and 219 true positives out of the expected 232 elements. The Acme Air reference system consists of five services and one package service that define common interfaces. For these, a total of 31 components, 33 interfaces, 31 provider roles, and 30 required roles are evaluated, for a total of 125 elements. The model extracted with the technology-specific rules has a total of 78 correctly identified elements, 47 missing elements, and 4 incorrect elements. With this new project-specific rule set, 30 items are still missing, but no items are incorrect, and 95 items are correct.

Table 1: Summary of results. Subscripts indicate the first run with technology-specific rules and the second run with additional project-specific rules. The number of rules (NuR) and LOC are given for the project-specific rules.

Case Study	p_1	r_1	$F1_1$	p_2	r_2	$F1_2$	NuR	LOC
Spring PetClinic	93.2%	93.2%	93.2%	-	-	-	-	-
Piggy Metrics	90.6%	82.3%	86.2%	91.2%	88.7%	89.9%	2	7
Tea Store	100.0%	44.8%	61.9%	93.6%	94.4%	94.0%	4	29
Acme Air	95.1%	62.4%	75.4%	100%	76.0%	86.4%	3	32

Results and Discussion: The summary of the evaluation results of the four case studies for the technology-specific and project-specific rules is shown in table 1. Precision values range from 90.6% to 100.0%, recall values from 44.8% to 94.4%, and $F1$-scores from 61.9% to 94.0%. For the Tea Store system, our approach achieves the best overall $F1$-score using the improved project-specific rules. The Spring PetClinic system performs best on the first pass with the technology-specific with a $F1$-score of 93.2% because it strictly follows the Spring framework patterns. The evaluation shows that general technology-specific rules have a higher risk of falsely classifying elements as relevant. However, the results generally improve when these rules are used together with other project-specific rules. On average, a new rule improves results by less than 2.0%. Piggy Metrics improved by 3.7% with two rules, while Tea Store improved by 32.1% and Acme Air improved by 11.0% with more rules. JAX-RS systems benefit more

[3] https://github.com/DescartesResearch/TeaStore
[4] https://github.com/acmeair/acmeair

from new rules than Spring-based systems due to fewer technology-specific implementation requirements and constraints. The results show that opinionated frameworks like Spring provide a better foundation for technology-specific rule sets than weaker frameworks. Comparing the LOC required for project-specific rule sets reveals differences between Spring and JAX-RS systems. JAX-RS systems required more LOC than the Piggy Metrics system because a single rule covers all discoverable elements. This highlights an area for future improvement to simplify rule implementation. A poor result from the rule engine may indicate a poorly designed system with missing patterns. To address this, architects can implement new project-specific rules. In this way, the rule engine can act as a warning system to check the quality of a relevant system when bad extraction results are given.

5 Related Work

Garcia et al. conclude that clustering of software entities is the almost universally used method for automated architecture reconstruction [7]. In most cases, a graph structure is generated based on dependencies in the source code, so that components can be reconstructed using clustering or pattern matching. Although each of these reverse engineering methods has a different principle, they all divide source code entities into mutually exclusive clusters, each based on a dominant principle such as cohesion and coupling or naming patterns. Garzón et al. propose an approach for reverse engineering object-oriented code into a unified language for both object-oriented programming and modeling [8]. Using an incremental and rule-based approach, UML class diagrams and state machines can be mixed with the associated source code. However, these rules cover only the basic object-oriented constructs, not specific technologies. Starting from the assumption that most well-designed systems follow strict architecture design rules, Cai et al. propose a new perspective for architecture reconstruction [3]. Their so-called ArchDRH clustering family allows design rule-based clustering to be combined with other clustering techniques to partition a large system into subsystems. However, these design rules take the form of special program constructs, like shared data structures or abstract interfaces, that are not used by any of the submodules. In their literature review, Raibulet et al. compare fifteen different model-driven reverse engineering approaches and find that the approaches and their application areas are versatile. In this respect, MoDisco [2] is the most related approach in a comprehensive scope [11]. Bruneliere et al. developed the generic and extensible MoDisco approach, which provides support for Java, JEE, and XML technologies to generate model-based views of the architecture. Although MoDisco is extensible with technologies, it does not support direct reuse of a technology's common concepts.

6 Conclusion

This paper presents a novel approach for building static system structure models in component-based software systems using reusable concept descriptions for

reverse engineering. The approach uses structural mapping rules to reconstruct models from source code, considering technology-specific relationships and concepts. The contributions of the approach include formally defined rules created by technology experts prior to the automatic extraction process, and a rule engine that can apply these rules to produce consistent software models. Evaluation of reference systems using Spring and JAX-RS technologies demonstrated the effectiveness of the approach. The evaluation also showed the potential to improve the rule system by integrating project-specific rules. The automatic model generation enabled by this approach has the potential to improve software maintainability and support architectural decisions in component-based software systems. Future work includes investigating the application of the approach to different types of systems, evaluating its scalability and efficiency, and developing a knowledge base of technology-specific rules to improve its reusability in similar projects.

References

1. Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd (2016)
2. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: Modisco: A generic and extensible framework for model driven reverse engineering. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (2010)
3. Cai, Y., Wang, H., Wong, S., Wang, L.: Leveraging design rules to improve software architecture recovery. In: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures. QoSA '13, ACM (2013)
4. Canfora, G., Di Penta, M.: New frontiers of reverse engineering. In: 2007 Future of Software Engineering. FOSE '07, IEEE Computer Society (2007)
5. Canfora, G., Di Penta, M., Cerulo, L.: Achievements and challenges in software reverse engineering. Commun. ACM (2011)
6. Favre, J.M.: Foundations of model (driven) (reverse) engineering. In: Language Engineering for Model-Driven Software Development (2005)
7. Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: ASE'13 (2013)
8. Garzón, M.A., Lethbridge, T.C., Aljamaan, H.I., Badreddin, O.: Reverse engineering of object-oriented code into umple using an incremental and rule-based approach. In: CASCON'14 (2014)
9. Kazman, R., Woods, S., Carriere, S.: Requirements for integrating software architecture and reengineering models: Corum ii. In: Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261) (1998)
10. Pautasso, C., Wilde, E.: Restful web services: Principles, patterns, emerging technologies. In: Proceedings of the 19th International Conference on World Wide Web. WWW '10, ACM (2010)
11. Raibulet, C., Fontana, F.A., Zanoni, M.: Model-driven reverse engineering approaches: A systematic literature review. IEEE Access (2017)
12. Reussner, R.H., Becker, S., Happe, J., Koziolek, A., Koziolek, H.: Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press (2016)
13. Roy, J., Ramanujan, A.: Understanding web services. IT Prof. (2001)