An Automated Approach to Recover the Use-case View of an Architecture

Joanna C. S. Santos
Rochester Institute of Technology
Rochester, NY. United States
jds5109@rit.edu

Sara Moshtari
Rochester Institute of Technology
Rochester, NY. United States
sm2481@rit.edu

Mehdi Mirakhorli
Rochester Institute of Technology
Rochester, NY. United States
mxmvse@rit.edu

***Abstract*—Many tools and techniques were described in the literature for automated recovery of software architecture from software artifacts, such as code. These approaches generate models of software architecture at different levels of granularity and notations. Whereas there is a vast literature in recovering components, packages, and interactions between them, we lack automated approaches for recovering the use-case view of an architecture. In the 4+1 view of the architecture, use-case view or scenarios represents a view of the architecture in terms of the systems' core functionalities provided to end-users. This view is essential in understanding the system and its underlying architectural decisions. Manually recovering and documenting the application's scenarios from source code is time-consuming, as large-scale enterprise systems can have a large number of scenarios. In this NEMI paper, we present a novel automated approach for recovering the scenarios from Web applications source code. These scenarios are shown in use case diagrams alongside with sequence diagrams that further describe how each use case is implemented in the system. Our approach works under the assumption that the URLs (endpoints) of a Web application can give us clues to the system's use cases. Therefore, our technique combines a set of heuristics and static analysis in order to detect the endpoints in a Java Web application as well as the backend classes and methods that will process the request. Subsequently, it uses Natural Language Processing (NLP) techniques to extract use cases from these identified endpoints and uses the computed program slices to generate sequence diagrams for each identified use case. We conducted an initial evaluation of our approach by detecting endpoints in Sagan, an existing open-source Web application. We then demonstrate the use cases generated and how their implementation looks like through sequence diagrams.**

***Index Terms*—Architecture recovery, Use case view, Web applications endpoints**

I. Introduction

Over the last decade, different tools and techniques have been proposed for reconstructing architecture from source code or other artifacts [1], [4], [10], [12], [13], [16]. This is mainly conducted to understand the program structure or architecture so that developers can modify the eroded system and restore it to the intended architecture, or renovate the architecture by changing it to a new optimal design.

A recovered architecture can be presented in different ways at different levels of formalisms (simple boxes and lines, or even formal languages). In the *4+1* model [8], an architecture is modeled using four views (logical, process, development and physical) plus one view of the architecture that governs all the other views. This "+1" view is the ***use-case view*** of the architecture and it models the system's main functionalities (i.e., the scenarios). As pointed by Kruchten, although one can omit a view (e.g., the physical view for smaller systems), the use-case view is always *crucial* because it is the driver for all the other views [8]. Despite this importance, to this date, we lack an automated approach that can recover scenarios from software artifacts to aid architects in communicating and understanding the underlying software's architecture.

One of the key challenges for automated recovery of the use-case view of a system's architecture is to extract *semantics* from software artifacts (whose abstraction level is at a much lower level compared to that handled by humans) in order to generate meaningful and understandable models. Previous work on obtaining semantics from code to guide architectural recovery mapped their recovered architecture to predefined views [6], [7] or required architects to write queries [11].

In this NEMI paper, we argue that the *endpoints* (i.e., the URLs that the application accepts HTTP requests) of Web applications can give clues on the use cases implemented by the system. As a result, they could be leveraged for the automated extraction of the use-case view of the architecture. This way, we tackle the challenge of extracting semantics from low-level software artifacts in order to help architects understand and evaluate the underlying architecture. In this context, we describe our early results in developing ***an automated technique to recover the use-case view of the architecture of Java Web applications*** by relying on the application's *endpoints*. Our approach first performs static analysis of the backend code and apply a set of framework-specific heuristics to find the application's ***endpoints specifications***. These specifications describe the following attributes for each endpoint: the Java classes that listen to that URL (*endpoint classes*); the methods from the classes that will actually process the request to that URL (*entrypoint methods*); any other non-endpoint classes that are involved in processing the request (*related backend classes*); and a sequence of statements that will be involved in processing the request (*program slices*). Subsequently, it applies Natural Language Processing (NLP) techniques [9] to infer the system's ***use cases***. Finally, we leverage the program slices previously computed to communicate *where* and *how* each use case is implemented via sequence diagrams. Therefore, the output of our technique is the **use-case view of the architecture** modeled in terms of *UML use case diagrams* alongside with *sequence diagrams* that further detail how each functionality is implemented in the code.

II. Approach Overview

Figure 1 shows an overview of the approach, whose steps are described in the next subsections.

A. Heuristic-Based Extraction of Endpoints Metadata

Under the assumption that an application's URL can point to the functionality provided by the system, our approach first extracts ***endpoints metadata*** from the application's source code[1]. An endpoint's metadata is the combination of its *URL*, the *entrypoint methods* that process the request to that URL and the *classes* to which these methods belong to.

Since the way an application implements the classes/methods that will accept an HTTP request varies according to the framework being used, we developed a set of heuristics to detect the endpoint's metadata. Table I lists the applied heuristics. We currently offer support to the detection of endpoints for applications that use the J2EE, Spring MVC or the GWT frameworks.

The first set of heuristics (**J1**-**J4**) abstracts the cases in which an application uses the standard Java Web API (*J2EE*). In this case, Web applications implement *Servlets* and *Filters* to process HTTP requests, which can be defined in an XML file or via class annotations. The heuristic **J5** is used to handle the case where server-side scripts embedded in JSP files are directly served to the client. The heuristics **S1**-**S4** are used to detect endpoints from applications that use *Spring MVC*, a Web framework that follows the Model-View-Controller architectural pattern. These heuristics detect the use of *handler mappings* of the framework to declare URLs. The last heuristic (**G1**) is used for supporting applications that used the Google Web Toolkit (*GWT*). This heuristic extracts URL declarations from module descriptors (an XML configuration file).

B. Extraction of Endpoints Specifications

Subsequently, we extract the ***endpoints specifications*** which enrich the metadata from the previous step with two more attributes: *related backend classes* (any other non-endpoint classes that are involved in processing the request), and *program slices* (a sequence of statements that will be executed to process the request to the URL). To compute these specifications, we performed a 0-1-CFA (context-insensitive instance-based) static analysis [5] to build a callgraph[2], whose root nodes are the *entrypoint methods* within the endpoints metadata previously extracted. From the callgraph, we then extract the system's Interprocedural Control-Flow Graph (ICFG) [14] and traverses it to collect all the set of instructions reachable by each entrypoint method (*program slices*) and the classes that contain them (*related classes*).

[1]We use the application's source code because some of the heuristics work by parsing code annotations that may not be retained in the compiled JAR.

[2]Although we could perform the static analysis on top of the source code, we use the JAR file for the pragmatic reason that we could generate a compiled bytecode that includes all the dependencies recursively.

C. Extraction of the Use Case View

From the endpoints specifications computed on the previous step, we apply NLP techniques to derive the ***use case view of the architecture***. This view is modeled as an *use case diagram* along with a set of *sequence diagrams* that indicates how each use case is currently implemented in the code.

1) Use Case Diagrams Generation: To generate a use case diagram, we first need to identify a unique set of use cases and select an appropriate name for it. Our goal is to derive use case names that are short imperative sentences that describe an action, i.e. phrases that start with a *verb* and followed by its *object* (e.g., "Create account" or "Export access logs to CSV."). For each endpoint in the specifications extracted in the previous step, we apply four different rules to find the use case names. Overall, each rule encompasses three steps:

(1) Create a sentence s from an element in the endpoint specification.
(2) Verify whether s is well-formed. To check for well-formedness, we perform POS (Part-of-Speech) tagging [15] (to identify nouns, verbs, etc) and dependency parsing [2] (to determine relationships between words) on top of the generated sentence. A sentence is then considered well-formed if the root node of its semantic graph [3] is a *verb* and the *object* for that verb is explicit.
(3) If the sentence is well-formed, we use it as the use case name and move forward to generate a use case name for the next endpoint. Otherwise, we proceed to the next rule to generate a use case name for the current endpoint.

Each rule is described in detail below.

• ***Rule #1***: We split the entrypoint method's name by special characters to obtain a list of *tokens*. We further split each of these tokens by camel case. Then we create a sentence by joining each token with blank spaces and capitalizing the first letter (e.g., `importTeamMembersFromGithub()` is converted to "Import team members from Github").

• ***Rule #2***: If the entrypoint method's name is just a standalone verb (e.g. "`show()`") we need to find its object (i.e. the receiver of the action expressed by the verb). We attempt to find this object by searching for a noun. We first search within the URL; if no noun is found, we perform the same attempt but from the class name that has the entrypoint method. Hence, we first tokenize the corresponding URL for the method (first by "/", then by camel case) and we iterate backward on this list of tokens to find a noun. If we found a noun, we generate the label by joining the method name and the newly found noun. If no noun could be found in the previous step, then we repeat this process but using the class name of the entrypoint method[3]. If a noun was found, then we create a label by concatenating the method's name and the noun.

• ***Rule #3***: We tokenize the classname (removing any suffixes) and construct a sentence from it (e.g., *AddAuthorServlet* is converted to "Add author" — the Servlet suffix is removed).

[3]We tokenize class name and remove its suffix (if any), then we search for a noun in a backward fashion

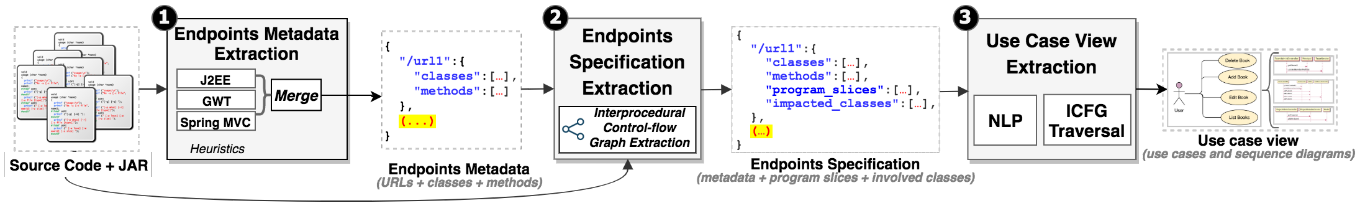
Source Code + JAR
1
Endpoints Metadata Extraction
J2EE
GWT
Spring MVC
Merge
Heuristics
{ "/url1":{ "classes":[...], "methods":[...] }, (...) }
Endpoints Metadata
(URLs + classes + methods)
2
Endpoints Specification Extraction
Interprocedural Control-flow Graph Extraction
{ "/url1":{ "classes":[...], "methods":[...], "program_slices":[...], "impacted_classes":[...], }, (...) }
Endpoints Specification
(metadata + program slices + involved classes)
3
Use Case View Extraction
NLP
ICFG Traversal
Delete Book
Add Book
Edit Book
List Books
User
Use case view
(use cases and sequence diagrams)

Fig. 1. Overview of our Automated Approach to Recover the Use-case View of an Architecture

• ***Rule #4***: If none of the rules above found a proper sentence, we create a use case label from the method name's sentence.

It is important to highlight that some imperative sentences are ambiguous during the POS tagging (e.g., the word "List" in the imperative sentence "List published posts" is misclassified as a noun rather than a verb). To tackle this problem we perform POS tagging on top of two versions of the generated sentence: one using the generated sentence as is, and another one by prepending the preposition "To" followed by the sentence (e.g., "To list published posts"). This helps the POS tagger to correctly disambiguate words that can be both nouns and verbs depending on its usage context.

To remove suffixes from class names in Rule #2 and #3, we get all the classes within the endpoint specifications and tokenize them by special characters and camel case. Subsequently, we construct a graph where the nodes are tokens and the edges represent the reverse order of the tokens in an identifier (e.g., "UserDAO" creates an edge from "DAO" to "User"). The suffixes are the *root nodes* in this graph, and we use this computed set of suffixes to clean class names.

By applying the rules above we obtain a list of tuples that have: *use case name*, its corresponding *endpoint* and *entry-point method*. Since real systems can have a large number of use cases, we group them by their ***architectural components***. To identify the components that a use case belongs to, we extract the top-most path in the endpoint's URL (e.g., the top-most part of the URL "/admin/blog" is "/admin").

2) Sequence Diagrams Generation: We traverse the program's Interprocedural Control-Flow Graph (ICFG) to derive a sequence diagram. We compute this diagram by stepping through the program's ICFG starting from the entrypoint method for the use case in order to list all method calls reachable from the entrypoint method. To prevent overwhelming developer with potentially large diagrams, we include application-only classes as participants in the sequence diagram (i.e., built-in Java classes or external APIs are disregarded).

III. Early Results

We demonstrate our technique using the **Sagan**[4] project, which is the code for the http://spring.io Web site.

• **RQ#1: Does our technique correctly extract all the application's endpoints?** In this research question, we evaluate whether we can correctly enumerate the endpoints within a Java Web application. To this end, one of the authors manually enumerated the endpoints (URLs) for Sagan to establish the ground truth. To ensure that this ground truth was trustworthy, we conducted a peer review in which we asked an external developer (with 3 years of experience) to scrutinize the accuracy of the manual dataset. Finally, we ran our approach against these projects and computed its precision and recall in detecting the Web endpoints.

[4]https://github.com/spring-io/sagan

TABLE I
Endpoint Detection Heuristics

J2EE Heuristics
J1: Using the <*servlet*> and <*servlet-mapping*> tags in a *web.xml* file
J2: Using the <*filter*> and <*filter-mapping*> tags in a *web.xml* file
J3: Using the @*WebServlet* class annotation
J4: Using the @*WebFilter* class annotation
J5: All public server-side scripts
Spring MVC Heuristics
S1: Using the @*Controller* and @*RequestMapping* annotations
S2: Using a Bean Name Url Handler Mapping defined in an XML file
S3: Using ControllerClassNameHandlerMapping defined in an XML file
S4: Using Simple Url Handler Mapping defined in an XML file
GWT Heuristics
G1: Using the Entrypoints declared in Module Descriptors

RQ#1 Results: Our approach detected a total of **66** endpoints. It achieved an **100% precision**, but it missed **7** endpoints for the Sagan project (**89% recall**). Our approach missed these endpoints because the application used custom class annotations to which our heuristics cannot detect.

• **RQ#2: Can our technique generate meaningful use case views?** In this question, we verify to which extent the automatically extracted use cases are meaningful for a software engineer. For this purpose, we gave to an external developer (the same developer as in RQ1) the generated Sagan architecture's use case view. For each extracted use case, we asked the developer the following question: *is the use case appropriate (i.e., do you consider that it provides a proper meaning to the system's functionality represented)*?

RQ#2 Results Figure 2 shows the use cases automatically identified and grouped by component. Figure 3 shows the sequence diagram for the use case "Delete post" (due to space constraints, we cannot show the sequence diagrams for all use cases). According to the developer, **41** out of these **57** (72%) generated use cases had names that provided proper meaning about the underlying system's functionality. The developer also pointed out that the use cases for the *"Admin, Badge, and Blog [components] communicate the most information"*, whereas the components *"Tools, Understanding, and Webhook do not provide much information. This is mostly due to the*

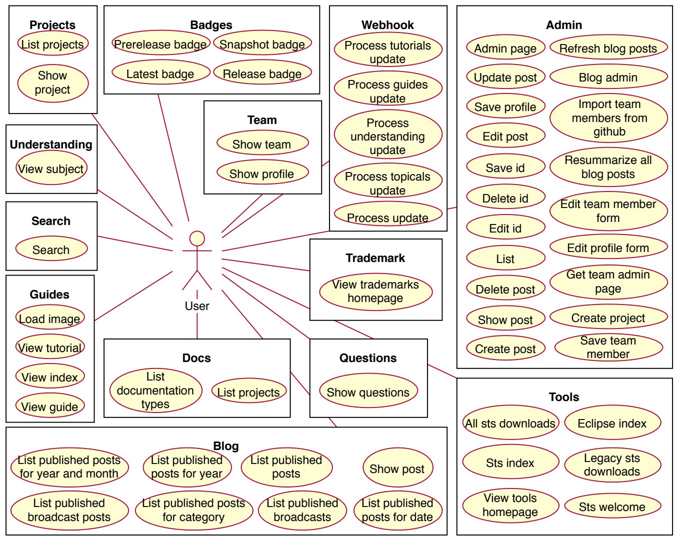

Fig. 2. UML Usecase Diagram Generated for the Case Study

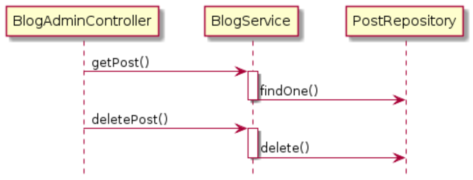

Fig. 3. UML Sequence Diagram Generated for the "Delete post" use case

descriptions being uninformative and lacking the context to get more information about what they mean (...)".

It is important to highlight that although the project had a total of 66 URLs, our approach generated 57 use cases since the heuristics described in Section II-C introduced duplicated names. During the diagram generation, we remove duplicated labels within the same component.

IV. Limitations

The main limitation of our approach is that it highly depends on whether developers provided meaningful identifiers for methods, classes and URLs. Moreover, the sequence diagrams are generated statically; and it is a well-known problem that static analysis may fail to properly compute the targets of method calls in face of polymorphism or dynamic programming language features (e.g., reflection) [14]. Lastly, our endpoint detection approach is currently tailored to Web applications developed using J2EE, SpringMVC, or GWT.

V. Conclusion and Future Work

In this NEMI paper, we explored the idea of leveraging a Web application's URLs (endpoints) to automatically infer the use-case view of its underlying architecture. Our main insight is that functionalities are usually separated into distinct URLs (which are hierarchical in nature) and, as a result, we could use them to guide the automated recovery of the use-case view of a system. We then demonstrated our early results in using this approach to recover the use-case view of an open-source Web application. We evaluated it in terms of precision and recall as well as by asking for feedback from a software developer. The current results are promising. As future work, we are investigating the use of stack traces and server logs to aid the process of uncovering endpoints and generating sequence diagrams from runtime information. We are also devising approaches to uncover actor types (e.g., User *vs* Admin) and relationships (between use cases, and actors).

Acknowledgments

This work was partially funded by the US National Science Foundation under grant numbers CNS-1816845 and CNS-1823246.

References

[1] N. Alshuqayran, N. Ali, and R. Evans. Towards micro service architecture recovery: An empirical study. In *2018 International Conference on Software Architecture (ICSA)*, pages 47–4709. IEEE, 2018.

[2] D. Chen and C. Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014.

[3] M.-C. De Marneffe, T. Grenager, B. MacCartney, D. Cer, D. Ramage, C. Kiddon, and C. D. Manning. Aligning semantic graphs for textual inference and machine reading. In *Proceedings of the AAAI Spring Symposium*, pages 468–476, 2007.

[4] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 486–496. IEEE Press, 2013.

[5] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.

[6] D. R. Harris, A. S. Yeh, and H. B. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3(1-2):109–138, 1996.

[7] V. Jakobac, N. Medvidovic, and A. Egyed. Separating architectural concerns to ease program understanding. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[8] P. Kruchten. Architectural blueprints–the 4+ 1 view model of software architecture, software, vol. 12, number 6, 1995.

[9] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[10] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic centric approach for automating traceability of quality concerns. In *International Conference on Software Engineering, ICSE (1)*, 2012.

[11] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 13–22. IEEE, 1999.

[12] B. R. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Trans. Software Eng.*, 32(7):454–466, 2006.

[13] M. Shtern and V. Tzerpos. Clustering methodologies for software engineering. *Adv. Soft. Eng.*, 2012, Jan. 2012.

[14] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

[15] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for computational Linguistics, 2003.

[16] Xinyi Dong and M. W. Godfrey. A hybrid program model for object-oriented reverse engineering. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 81–90, June 2007.