Spotlighting Use Case Specific Architectures

Mohamed Lamine Kerdoudi, Mohamed Lamine Kerdoudi, Chouki Tibermacine, Salah Sadou

Spotlighting Use Case Specific Architectures

Mohamed Lamine Kerdoudi[1,2], Chouki Tibermacine[2], and Salah Sadou[3]

[1] Computer Science Department, University of Biskra, Algeria
[2] LIRMM, CNRS and Montpellier University, France
[3] IRISA- University of South Brittany, France
lamine.kerdoudi@gmail.com, Chouki.Tibermacine@lirmm.fr,
Salah.Sadou@irisa.fr

Abstract. Most of the time a large software system implies a complex architecture. However, at some point of the system's execution, its components are not necessarily all running. Indeed, some components may not be concerned by a given use case, and therefore they do not consume/use or register the declared services. Thus, these architectural elements (components and their services) represent a "noise" in the architecture model of the system. Their elimination from the architecture model may greatly reduce its complexity, and consequently helps developers in their maintenance tasks. In our work, we argue that a large service-oriented system has, not only one, but several architectures, which are specific to its runtime use cases. Indeed, each architecture reflects the services, and thereby the components, which are really useful for a given use case. In this paper, we present an approach for recovering such use case specific architectures of service-oriented systems. Architectures are recovered both through a source code analysis and by querying the runtime environment and the service registry. The first built architecture (the core architecture) is composed of the components that are present in all the use cases. Then, depending on a particular use case, this core architecture will be enriched with only the needed components.

1 Introduction

The context of this work is the architecture of large-sized service-oriented software systems. By large-sized systems, we mean systems that are composed of hundreds to thousands of components, registering and consuming hundreds of services. Architectures of systems in general are important to be explicitly modeled, and this is particularly critical for large systems. When such architecture models are not explicit, it becomes important to recover them from the system's artifacts (e.g., source code). Architecture recovery is a challenging problem, and several works in the literature have already proposed contributions to solve it (e.g., works cited in [8, 13, 15]). Architectures recovered from large systems are however complex and difficult to "grasp". Indeed, architectures of large systems model a lot of components, their contracts (required and provided interfaces) and their numerous and tangled interconnections. If we add, to these architecture elements, services that are registered and consumed by components (which enrich their contracts), these architectures can be easily assimilated to "spaghetti" code.

We noticed that at some point in the execution of such large systems, not all their components are running/active. Components that are not running and their properties (services and their connections) represent a "noise" in a recovered (complex –"spaghetti") architecture. Their elimination reduces thereby the complexity of this architecture and helps the developers in their maintenance tasks. In this work, we argue that large systems do not have a single large and complex architecture, but rather several architectures depending on the use context. In this paper, we present an approach (Section 2) which enables to recover the architecture of a service-oriented system, depending on a particular use case. This approach contributes with a process that analyzes the source code of the system and interacts with the runtime environment, including the service registry, to build a first core architecture modeling the components of the system that always run. Then, this core architecture is enriched with new elements that reify the runtime entities involved in a particular use case, of interest for the developer (in which a bug occured, for instance). Simplifying architecture models in this way enables developers to make like a quick "inventory" of what is concretely running, among all what composes their system, at a particular execution time. They can easily identify which component is consuming a particular failing service, for instance. In the literature there is no efficient process for recovering these dynamic use case architectures from running systems (see Section 5).

We implemented the proposed process for the OSGi platform (see Section 3) and we experimented it on a set of real-world Eclipse-based applications (see Section 4). At the end of the paper, we highlight the interests and limitations of the proposed process, as well as some future directions of this work (Section 6).

2 General Approach

The problem with traditional architectural models of a software system is that they describe all involved components and their potential dependencies. The proposed process (see Figure 1) enables to produce an architecture model that can be used by the developer to solve a maintenance problem related to a given use case. First, we create the core architecture, which represents only components that exist in the system whatever the executed application's use case. In the second step, we use traces obtained by executing scenarios corresponding to the application's use cases to identify what we call "use case"-specific (or use-case) architectures. The latter are built around the core architecture with variants (adding new components, services, etc.) concerning the executed use case.

Recovering the Core Architecture : To create the core architecture, we use first a static analysis to collect all the components involved at the system's starting time. The core architecture will be comprehensive once the dynamic elements are identified. Indeed, some dependencies exist only through requests for services made during execution time. To identify these dependencies, we launch the application without applying a use case ("Use Case 0" in Figure 1).

Recovering Use Case Architectures : During a maintenance activity, the developer focuses on a given use case of the application. Thus, we ask a developer to execute a set of use cases and we capture all traces produced by the involved

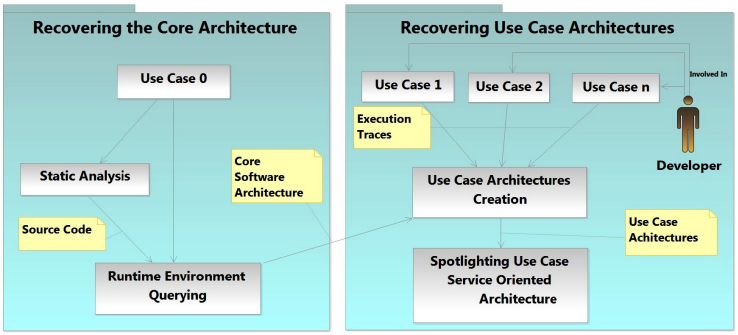

Fig. 1: Proposed Approach

components. After that, we parse the code of the newly activated components in order to identify their dependencies. The collected information is used to enrich the core architecture in order to build the "use case"-specific architecture.

3 Implementation of the Approach: Case of OSGi

We implemented our approach for OSGi-based systems. OSGi is a specification that defines a component model and a framework for creating highly modular Java systems[16]. An OSGi component is known as a bundle that packages a subset of the Java classes, and a manifest file. The OSGi framework introduces a service-oriented programming model. Indeed, a bundle (provider) can publish its services into a Service Registry, while another bundle searches the registry to use available services. We take as a running example an Eclipse-based application that runs on top of Equinox, which is the reference implementation of the OSGi specification. We used the release: Eclipse JEE for Web Developers, Oxygen.2[1].

3.1 Recovering the Eclipse Core Architecture

In order to recover the core architecture of the Eclipse-based application, we first perform a static analysis of the source code and the manifest files of the bundles that are needed to start this application. These bundles refer to components that have the state "ACTIVE". They are recognized by querying the runtime environment. Indeed, we have added listeners in the Eclipse plugin which implements the proposed process. We rely on SCA[2] for the modeling of the obtained architecture. SCA has been chosen because of its simplicity and the existence of good tools support for the graphical visualization. First, each bundle is modeled as an SCA component which has as a name the bundle's symbolic name. Then, by parsing the manifest files, we identify the dependencies between components. Indeed, we consider each declared interface in the exported package as a provided interface and the declared interfaces in the imported packages are considered as required interfaces. The SCA Wires are used to represent the connections. After that, we hide the required interfaces that are not concretely used in code.

[1] Downloaded from repository: https://lc.cx/P2Qw
[2] SCA is a set of specifications which describe SOA systems: https://lc.cx/AEP3.

Besides, in the context of OSGi components, services are defined by dedicated classes that are instantiated and registered with the OSGi Service Registry either programmatically or declaratively (i.e., using the OSGi DS framework). Services declared with DS framework are identified by parsing the "$OSGI-INF/component.xml$" files. For the programmatically registered services, we parse the following two statements: `<context>.registerService(..)` and `<context>.getServiceReference(..)`. Then, the core architecture is enriched by dynamic features. Indeed, we query at runtime the execution environment and Service Registry to identify what are the concretely registered dynamic services and consumed services. Therefore, we hide the static information.

3.2 Recovering Eclipse Use Case Architectures

Once the core architecture is recovered, we ask the developer to execute a set of scenarios corresponding to use cases. New components related to each scenario can be activated and new services can be registered. These components and services, are identified by querying at runtime the execution environment and the Service Registry. As consequence, for each scenario, we generate a runtime use case architecture by adding to the core architecture the newly activated components, interfaces, and services. For instance, after executing the following use case: "Accessing the Toolbar Menu, Opening Help− >Install New Software...", 11 new components are activated. Figure 2 shows an excerpt of the recovered use case architecture for this scenario. We show in this figure the new activated components (surrounded by bold lines) which are connected to the core architecture components. For reasons of readability, we show only some core architecture elements that are directly connected to the newly activated components.

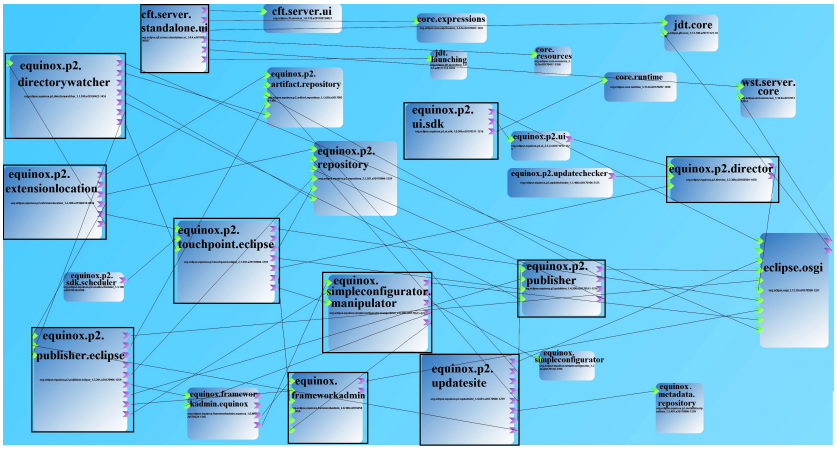

Fig. 2: A "Use Case"-specific Architecture

Besides, we offer also to the developers a way to refine the recovered use case architecture and spotlight the implicit service-oriented architecture (pure SOA), which contains only services (without interfaces) and the active components that register or consume services. In this way, we enable them to focus only on services-based dependencies, which simplify greatly the architecture model.

4 Empirical Evaluation

We evaluated our approach starting from two Eclipse-based applications of different sizes. The aim is to measure the gain in the reduction of complexity of the recovered runtime use case architectures. Indeed, we have compared the complexity of the architecture that is obtained by a static code analysis of all the system components with the complexity of the recovered use case architectures. Table 1 describes the chosen systems[3]. For each system, we executed 4 use cases related to the installed projects. In order to measure the complexity of the recovered architectures, we have used a complexity metric (CM) proposed in [10]: $CM = \frac{AC}{AC_w}$, where, AC is Absolute Complexity of a use case architecture and AC_w is the worst architecture complexity which corresponds to the static architecture complexity. To estimate AC, we create an adjacency matrix from the architecture and we calculate the influence degree of each component on the rest of the system.

Table 1: Selected Eclipse-Based Applications

S. Id.	description	installed projects	# of bundles	# of classes	SLOC
1	Eclipse JEE for Web Developers Oxygen.2	**WTP**, **BPEL**, **Axis Tools**.	**1040**	**131282**	**4.11M**
2	Eclipse Modeling Tools Oxygen.2	**ArchStudio**, **Papyrus**, **BPMN2**.	**1502**	**151471**	**4.90M**

4.1 Complexity Measurement Results

The obtained results are presented in Table 2. As we can see in column 2, the static architectures of the two candidate systems are very complex and this is particularly true for the largest application. Column 4 presents the number of actions on the graphical user interface in order to describe quantitatively each use case. We can see (in Column 5) that the complexity of all the obtained use case architectures is greatly less than the complexity of the static architectures (AC_w). This confirms our intuition that focusing on the runtime use case architectures greatly reduces the complexity of the architecture compared to the static one. Second, the obtained CM values (Column 6) are good for all the recovered use case architectures. However, we noticed that these values decrease when we increase the size of the system. If we take UC2 in the two systems,

Table 2: Experiment Results

S. Id.	AC_w	Use Case	# of GUI Actions	AC	CM	# of Active Components
1	**5637**	UC 0	0	**1076**	**0.19**	**163**
		UC 1	11	**1195**	**0.21**	**174**
		UC 2	28	**1777**	**0.31**	**242**
		UC 3	35	**1907**	**0.33**	**248**
		UC 4	55	**1941**	**0.34**	**259**
2	**9014**	UC 0	0	**2153**	**0.23**	**392**
		UC 1	4	**2197**	**0.24**	**394**
		UC 2	27	**2330**	**0.25**	**413**
		UC 3	30	**2429**	**0.26**	**425**
		UC 4	49	**2885**	**0.32**	**473**

[3] They have been downloaded from the following repository: https://lc.cx/m77k

which have almost equal number of GUI actions, we can see that CM value in the second system is less than in the first system (0.25 vs. 0.31). This because, the AC_w increases with the system size, while the AC vary in a stable interval. Third, we can observe in column 7 that the average number of newly activated components is equal to 50 components per use case. This can be considered as a good value for a system that contains more than a thousand components. Developers recover and understand the core architecture once (it is common to all use cases), which is considered as the initial overhead of our approach. After that, they can focus only on the newly activated components for a use case. At the end, we can observe the high correlation between the number of GUI actions and CM values (correlation coefficient equal to 0.86 for System 1 and 0.88 for System 2). The more the GUI actions we do, the greater CM values we obtain. But CM values remain very low, AC is thereby kept far below AC_w.

4.2 Performance Measurement

We evaluated the performance of our approach by estimating the time for recovering each architecture. We ran our experiments in a machine with a CPU 4.20GHz Intel Core i7-7700K, with 8 logical cores, 4 physical cores, and 32 GB of memory. The recovering of the static architectures takes 4 hours for the first System and 9 hours for the second System. Besides, the average time for recovering a use case architecture is 45 minutes for the first System and 2 hours for the second System. Therefore, this results demonstrate the efficiency our approach.

4.3 Threats to Validity

This experiment may suffer from some threats to the validity of its results:

Internal validity In order to evaluate the accuracy of our approach, we need to compare the recovered architectures with "ground-truth" use case architectures. A "ground-truth" architecture is an architecture that has been verified as accurate by the architects [9]. Obtaining this architecture is challenging. To mitigate this threat, we have verified manually the component dependencies of large parts of the recovered use case architectures by analyzing and checking manually source code and the manifest files of the candidate components.

External validity Our evaluation is based on set of OSGi systems which limits our study's generalizability to other kind of systems. To mitigate this threat, we selected systems providing different functionalities (BPMN, BPEL,...) and sizes.

5 Related Work

A framework comprising a set of principles and processes for recovering systems' ground-truth architectures has been proposed in [9, 13]. The authors in [15] provide a review of the hierarchical clustering techniques which seeks to build a hierarchy of clusters starting from implementation level entities. In our work, we focus on runtime use case architectures, instead of recovering whole static architectures. However, if the recovered use case architectures remain complex for a human analysis, we can use of one of the existing clustring methods for abstracting those architectures. The works in [5, 21, 4, 3] focused on extracting component-based architectures from existing object-oriented systems. Seriai et

al. in [20] used FCA to identify the component interfaces. Unlike these works, in our work, we deal with reducing the size of the recovered architecture by focusing on particular use cases, and we include dynamic features in this architecture.

Besides, several SOA recovery approaches have been proposed in the literature as part of the process of migrating systems to SOA solutions [18]. Most of these approaches are based on static code analysis of the target system. Examples of these works are [17, 2, 11]. A number of works such as [7, 22, 12] have been proposed to detect SOA patterns from service oriented applications. Our approach focused on the recovery of pure SOAs. Using SOA design patterns may be a good complement to our approach for a better understanding of the recovered architecture. More particularly, this helps in better understanding the design decisions made during the modeling of the analyzed system.

Managing complex architectures of large software systems became a topic of interest of several research works. Some authors proposed to organize architectural information using a Dependency Structure Matrix [19, 14]. The authors in [6] have proposed an architectural slicing and abstraction approach for reducing the model complexity. Abi-Antoun et al. [1] proposed a technique to statically extract a hierarchical runtime architecture from object-oriented code. In our approach, we deal with architectures at a higher level of granularity (component ones) and not low level ones (at object-oriented program level).

6 Conclusion and Future Work

In this work, we noticed that recovering the whole architecture of a large system produces models that are not tractable for developers due to their size and complexity. In this paper, we proposed a process for recovering the architecture of large component-/service-oriented systems. Since services in these systems are not provided and consumed all together, in a given use case, and components are not all active in the same time, we defined in this process a method to reduce the size and the complexity of the architecture. Thanks to a runtime analysis and taking into consideration only specific use cases of interest for the developer (related to a bug occurrence, for instance), we spotlight the active elements (components and services) in the recovered architecture. We benefited from the OSGi framework capabilities to implement such a process, and we experimented it on a set of Eclipse-based applications. The results showed the potential of the approach in recovering the architectures of these large systems, while reducing their complexity by spotlighting essential elements.

As a future work, we plan to make the recovered architecture models dynamic: they evolve (elements are shown and hidden) while the system is running by following debugger-like behaviors. In this way, we help the developer to monitor and evolve her/his system directly via its architecture. In addition, we want to make them interactive, by enabling developers to control components and services just by clicking, dragging and dropping the visualized elements.

References

1. Abi-Antoun, M., Aldrich, J.: Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In: Proc. of the ACM

OOPSLA (2009)
2. Alahmari, S., Zaluska, E., De Roure, D.: A service identification framework for legacy system migration into soa. In: Proc. of the IEEE SCC 2010. IEEE (2010)
3. Allier, S., Sadou, S., Sahraoui, H.A., Fleurquin, R.: From object-oriented applications to component-oriented applications via component- oriented architecture. In: Proc. of WICSA, Colorado, USA. IEEE (2011)
4. Allier, S., Sahraoui, H.A., Sadou, S., Vaucher, S.: Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In: Proc. of the 13th CBSE'10. pp. 216–231. Springer (2010)
5. Chardigny, S., Seriai, A., Oussalah, M., Tamzalit, D.: Extraction of component-based architecture from object-oriented systems. In: Proc. of WICSA. IEEE (2008)
6. Colangelo, D., Compare, D., Inverardi, P., Pelliccione, P.: Reducing software architecture models complexity: A slicing and abstraction approach. In: Proc. of FORTE'06. Springer
7. Demange, A., Moha, N., Tremblay, G.: Detection of soa patterns. In: Proc. of the 11th ICSOC'13. Springer (2013)
8. Ducasse, S., Pollet, D.: Software architecture reconstruction: A process-oriented taxonomy. IEEE TSE 35(4), 573–591 (2009)
9. Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: Proc. of IEEE/ACM ASE (2013)
10. Jiao, F., Hu, C., Zhao, C.: A software complexity metric for sca specification. In: Proc. of the CSSE. IEEE (2008)
11. Kerdoudi, M.L., Tibermacine, C., Sadou, S.: Opening web applications for third-party development: a service-oriented solution. Journal of SOCA 10(4), 437–463 (2016)
12. Liang, Q.A., Chung, J.Y., Miller, S., Ouyang, Y.: Service pattern discovery of web service mining in web service registry-repository. In: Proc. of ICEBE'06 (2006)
13. Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., Kroeger, R.: Measuring the impact of code dependencies on software architecture recovery techniques. IEEE TSE 44(2), 159–181 (2018)
14. MacCormack, A., Rusnak, J., Baldwin, C.Y.: Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Management Science 52(7), 1015–1030 (2006)
15. Maqbool, O., Babri, H.: Hierarchical clustering for software architecture recovery. IEEE TSE 33(11) (2007)
16. McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox: Creating highly modular Java systems. Addison-Wesley Professional (2010)
17. O'Brien, L., Smith, D., Lewis, G.: Supporting migration to services using software architecture reconstruction. In: Proc. of STEP. IEEE (2005)
18. Razavian, M., Lago, P.: A systematic literature review on soa migration. Journal of Software: Evolution and Process 27(5), 337–372 (2015)
19. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: Proc. of the ACM OOPSLA. ACM (2005)
20. Seriai, A., Sadou, S., Sahraoui, H., Hamza, S.: Deriving component interfaces after a restructuring of a legacy system. In: Proc. of WICSA. IEEE (2014)
21. Seriai, A., Sadou, S., Sahraoui, H.A.: Enactment of components extracted from an object- oriented application. In: Proc. ECSA. Springer (2014)
22. Upadhyaya, B., Tang, R., Zou, Y.: An approach for mining service composition patterns from execution logs. Journal of Software: Evolution and Process 25(8), 841–870 (2013)