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 [9, 16, 18]). 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 software systems, not all their components are running/active. Components that are not running and their properties (provided, published or consumed 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 that reflects the use context. This approach contributes with a multi-step 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 (e.g., bug occurrence). They can easily identify which component is consuming a particular failing service, for instance. In the literature and practice of service-oriented computing and software engineering, there is no efficient process for recovering these dynamic use case architectures from running systems (see related works in Section 5).

We implemented the proposed process for the OSGi platform, which provides a well-know service-based framework for Java applications. This implementation is discussed in Section 3. We experimented the process on a set of real-world Eclipse-based applications. The results of this experimentation demonstrated the efficiency of the proposed process (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 approach we propose aims at recovering from the system the architecture model corresponding to a given use case. In such an architecture model, elements that are not necessary to the use case are not represented and therefore facilitate the understanding of the architecture.

Thus, the proposed process to implement our approach (see Figure 1) enables to produce an architecture model that can be used by the developer to understand the architecture of a large software system for solving a given maintenance problem related to a particular use case. In the first step of the process, we create the core architecture of the system, which represents only the needed components, and their dependencies, to start the system. These components ex-

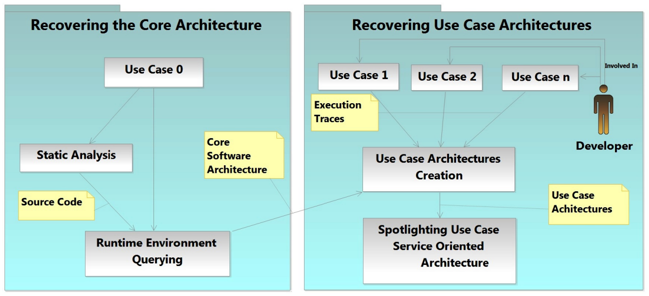
Recovering the Core Architecture
Use Case 0
Static Analysis
Core Software Architecture
Source Code
Runtime Environment Querying
Recovering Use Case Architectures
Use Case 1
Use Case 2
Use Case n
Involved In
Execution Traces
Developer
Use Case Architectures Creation
Use Case Achitectures
Spotlighting Use Case Service Oriented Architecture

Fig. 1: Proposed Approach

ist in the system architecture 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 architectures. The latter are built around the core architecture with variants (adding new components, services, interfaces, etc) concerning the executed use case. Below we give further details about these two constituent steps of our approach.

2.1 Recovering the Core Architecture

In any software system, there are components and some of their dependencies that are necessary for all use cases. As pointed above, we call the core architecture the architecture representing only these elements. Recovering the core architecture of the system is not only necessary to facilitate the construction of architectures corresponding to the different use cases, but also necessary to facilitate the identification of similarities and differences between the use cases. Some of the elements of the core architecture are statically identified while others are dynamically obtained. The formers require an analysis of the source code (static analysis), while the latters require an analysis of the execution traces (dynamic analysis).

By using a static analysis we aim first to collect all the components involved at the system's starting time. Components interact with each other through provided and required interfaces that represent the dependencies between them. We can identify the provided and required interfaces by parsing the description files and the meta-data (e.g. IDL files for CORBA components, Manifest files for OSGi components, or module descriptors for Java 9). However, the required interfaces that are declared in these files are not necessarily all effectively used by the component. Thus, we perform a static code analysis on each component to identify the unused required interfaces in order to hide them. Actually, there are two kinds of code dependencies: *include* dependencies and *symbol* dependencies [16]. The *include* dependencies established when one file declares that it includes another file (e.g. "*#include <foo.h>*" in C or "*import java.io.File;*" in Java), but does not use any function or variable declared in the included file. Using include dependencies, has no actual code dependency. In our approach, we use *symbol* dependency, which is more accurate. A symbol can be a function or a variable name. For example, if a method mA (declared in Component A) invokes a method mB (declared in Component B), then component A depends on Component B.

In addition to the identified provided/required interfaces, the components can also publish services in the Service Registry. Other components running in the same environment can then find and use those services. This relationship between components is considered as another kind of component dependencies. We identify these services by the static parsing of the component code.

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. In Figure 1, we characterized this by the "Use Case 0". Thus, we capture the dependencies that become effective at runtime. In addition to these dependencies, through them we can identify new components that will be added to the already constructed architecture.

2.2 Recovering Use Case Architectures

During a maintenance activity, the developer focuses on a given context of use of the application, which corresponds to one of its use cases. A use case describes the interaction of a user with a sequence of functionalities provided by the system and which gives a visible result for her/him. During the execution of a given use case, we capture all traces produced by the involved components (those already active and those that have just been activated). After that, we parse the source code of the newly activated components in order to identify their required and provided interfaces. We also take into account all static dependencies identified, during the first step, related to these newly activated components. The collected information is used to enrich the core architecture in order to build what we called the "use case"-specific (or use-case) 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 [12]. A component in OSGi is known as a bundle that packages a subset of the Java classes of the system, and a manifest file. The latter declares which of the packages are externally visible using "export-package". It also declares explicitly the bundles it depends on, using "import-package" or "require-bundle". The OSGi framework introduces a service-oriented programming model. Indeed, a bundle (service provider) can publish its services into a Service Registry, while a service client (another bundle) searches the registry to find available services to use. This bundle receives from the Service Registry a reference of a service implementation instance.

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 following release: Eclipse JEE for Web Developers, Oxygen.2 Release (4.7.2 –2017) [4]. This release contains 1040 bundles. The static architecture that we obtained by a static code analysis of all the components (active and inactive) of this Eclipse application is very complex and not helpful for a maintenance developer. This architecture should be more complex if it is reconstructed from a larger Eclipse application (more then thousands bundles) which is frequently the case in real development settings. We show in the next sections how to apply our optimizations in order to simplify and reduce the complexity of this architecture.

3.1 Recovering the Eclipse Core Architecture

We show here how the core architecture of Eclipse-based applications is obtained using the proposed process. In the first step we perform a static analysis of the source code and the manifest files of the OSGi components that are needed to start the Eclipse application. These bundles refer to the OSGi bundles that have the state "ACTIVE" during the Eclipse starting. 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[5] 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 of the architectures. First, each component (bundle) is modeled as an SCA component which has as a name the bundle's symbolic name[6]. Then, by parsing the manifest files of these components, we identify the dependencies between them. Indeed, we consider each declared interface in the exported package as a provided interface (modeled by an SCA Component Service, while its name corresponds to the qualified name of this interface) and the declared interfaces in the imported packages (and in the exported packages of the required bundles) are considered as required interfaces (modeled by SCA Component References). The SCA Wires are used to represent the connections between required and provided interfaces. After that, we hide the required interfaces that are not concretely used in code.

Once all interfaces and wires are created, we abstract away detailed connections between components in order to simplify the visualized dependencies between components. Indeed, there can be multiple dependencies between two components. Thus, if a component A requires several interfaces of the same component B, we will have one SCA Wire between the two components, and the names of the corresponding interfaces are hidden as documentation elements added to this SCA Wire element.

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.,

[4] Downloaded from repository: https://lc.cx/P2Qw

[5] SCA is a set of specifications which describe a model for building systems having a Service Oriented-Architecture : https://lc.cx/AEP3.

[6] It is a unique name which serves as an identity of the bundle in the whole system.

using the OSGi Declarative Services (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(...)`, in order to identify the registered service class name and its corresponding interface. The identified provided and consumed services are modeled respectively by SCA Component Service and SCA Component Reference. The SCA Wires are used to represent the connections between components consuming and providing these services.

In order to distinguish the representation of the required/provided interfaces and the services that are consumed/provided via the Service Registry, we use in the current implementation a specific naming convention for the services. The naming of the provided/consumed services follows the syntax: "`ServiceName_SR`". The `ServiceName` is obtained by parsing the parameters of the `context.registerService(...)` statements. Their types are deduced from the parsed code. For instance, in the Apache Felix Gogo bundles, the service `org.apache.felix.service.command.Converter` is provided by component: `org.eclipse.equinox.console`. This service is consumed by the `org.apache.felix.gogo.runtime` component, while this latter provides also an interface with the same name. The name of provided service via Service Registry (by the `equinox.console` component) is suffixed by "`_SR`".

After the construction of the core architecture based on the static analysis of the active components, the core architecture is enriched by dynamic service-oriented features. To do so, we query at runtime the execution environment and the Service Registry to identify what are the concretely registered dynamic services and consumed services. Therefore, the static information is then hidden from this architecture.

The recovered core architecture from the chosen Eclipse application enables us to model the components that always run in this system. This core architecture contains only 163 components which are needed to start this version of Eclipse. This makes it simple and may be very helpful for a maintenance developer compared to the whole static Eclipse architecture.

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 (event listeners of the implementation are executed, which generate traces). As consequence, for each executed scenario, we generate a runtime "use case"-specific architecture by adding to the core architecture the newly activated components, interfaces, and services. For instance, we have executed the following two use cases:

1. Accessing the Toolbar Menu, opening Help− >Install New Software...

2. Creating a BPEL Project, creating a BPEL Process Model and adding activities.

After executing the first use case, 11 new components are activated and added to the core architecture. 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.

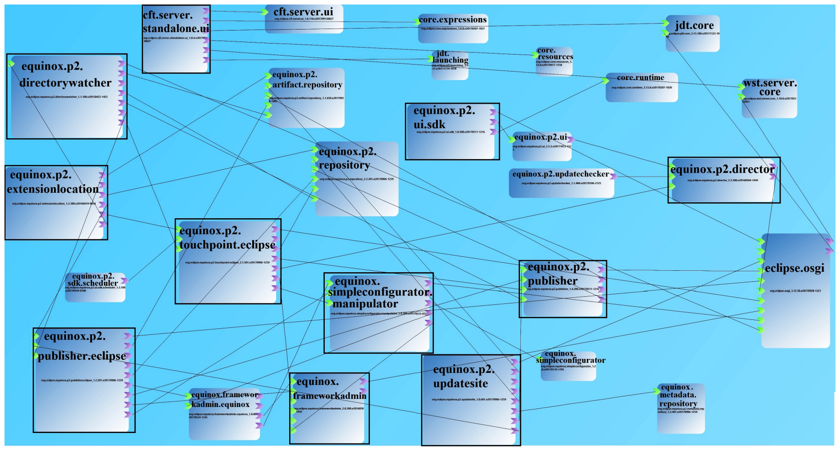

Fig. 2: A "Use Case"-specific Architecture

For the second use case, 67 new components are activated which is much greater than the number of activated components in the first use case. This is because the second use case is more complex and requires executing a higher number of actions. Still, the number of components in the two use cases is negligible compared to the total number of components in this system.

Besides, in this step, we offer to the developers a way to refine the recovered use case architecture and spotlight the runtime implicit service-oriented architecture, 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 conducted several experiments to evaluate our approach starting from two Eclipse-based applications of different sizes. The aim of these experiments is to measure the gain in the reduction of complexity of the recovered runtime "use case"-specific architectures using our approach. Therefore, we have addressed the following research question:

RQ: *To what extent the complexity of "use case"-specific architectures recovered using our approach is less than the complexity of the static architecture?*

For answering the research question, we measured first the complexity of the architecture obtained by a static code analysis of the candidate systems. Second, we used our approach to recover the core architecture and the runtime "use case"-specific architectures corresponding to a set of use cases. After that, we measured the complexity values of the obtained use case architectures, which are then compared with the complexity of the static architecture.

We choose Eclipse in our experiment because it is one of the largest and mostly used service/component-based Java system. A whole Eclipse release is considered as a component which is composed of several other components. This architectural vision makes it a modular and a scalable framework. The components are open for extension and configuration by third party developers.

Table 1 describes the chosen Eclipse-based systems[7]. In this table, we show the name of the system releases (in column 2) and the installed projects (in column 3), the number of bundles (in columns 4), the number of classes (in columns 5), and the application size in terms of number of source lines of code (in columns 6). For each eclipse release, we installed different projects such as, ArchStudio, Papyrus, and BPEL Project. This allows us to select different use cases related to the installed projects and compare the complexity of the use case architectures that are recovered from different kinds of applications.

Table 1: Selected Eclipse-Based Applications

S. Id	**description**	**installed projects**	**# of bundles**	**# of classes**	**SLOC**
1	Eclipse JEE for Web Developers Oxygen.2 R. (4.7.2)(2017)	**Web Tools Platform**, **BPEL Project**, **Axis Tools**.	**1040**	**131282**	**4.11M**
2	Eclipse Modeling Tools Oxygen.2 R. (4.7.2) (2017)	**ArchStudio 5.0.2**, **Papyrus 3.3.0**, **BPMN2 Modeler**.	**1502**	**151471**	**4.9M**

Here we describe the selected use cases that are executed in our evaluation:

– System 1:
  - UC 1: Accessing the Toolbar Menu and opening Help− >Install new Software...
  - UC 2: Creating a BPEL Project, creating a new BPEL Process by composing the created services and adding BPEL activities (Pick, Assign,...)
  - UC 3: Importing an existing Dynamic Web Project, adding the Apache Tomcat as a new Server Runtime, and running the Web project on Server
  - UC 4: Creating a new Dynamic Web Project, creating and editing new Java classes, creating and editing new HTML and JSP files
  - UC 5: Adding the Apache Axis2 as a Web service Runtime and the Apache Tomcat as a Server Runtime, importing an existing Web Project, creating bottom-up Web services, deploying and testing the Web services

[7] They have been downloaded from the following repository: https://www.eclipse.org/downloads/packages/

– System 2:
  - UC 1: Accessing the Toolbar Menu, opening Search– >File Search, and filling the file name and clicking on the search button
  - UC 2: Creating an Empty EMF Project, creating and editing an Ecore Model
  - UC 3: Creating BPMN2 Models, adding different BPMN elements (Lanes, Tasks, Gateways, ...)
  - UC 4: Creating a new ArchStudio Architecture Description, new State-charts, add different States and Transitions, opening the models in the Archipelago graphical Editor
  - UC 5: Creating Papyrus Project and creating Papyrus Models (component, activity, class diagrams)

As we can see, we have selected simple and complex use cases, which allowed us to recover "use case"-specific architectures of different sizes and complexities. In fact, the same use case can be executed in different ways, by performing different sequence of actions. Thus, we performed several runs per use case in order to ensure that all the runs have activated the same components. This allows to avoid errors (e.g., clicking on the wrong buttons) that can be made by the developer during the use case execution. We keep only the use case architecture that contains the activated components that are common to all the runs.

4.1 Complexity Measurement

In order to measure the complexity of the recovered software architectures, we have used a complexity metric based on SCA specification which is proposed in [13]. This metric is composed of three parts that are employed to measure the complexity of a component, a dependency, and a composite. The architectures recovered in our work are SCA composites. Thus, the following complexity metric (CM) is used for a composite:

$$CM = \frac{AC}{AC_w} \tag{1}$$

– AC is Absolute Complexity of a composite
– AC_w is the worst architecture complexity value,

In our experimentation, the AC_w corresponds to the absolute complexity AC of the recovered architecture by static analysis, which is considered as the "worst" case. Now, if $CM = 1$ for a runtime "use case"-specific architecture, this means that this architecture is very complex, like the worst architecture.

An SCA composite contains components and dependencies which can be modeled using a Weighted Dependency Graph (WDG). To calculate the absolute complexity (AC) of this composite, we proceed as follows:

1. First, we create the adjacency matrix $A\_Matrix_{n\times n}$ from WDG. Each element d_{ij} is specified as:

$$d_{ij} = \begin{cases} d(c_i), i = j \\ d(e_{ij}), i \neq j \wedge \exists e_{ij} \in E \\ 0, i \neq j \wedge \nexists e_{ij} \in E \end{cases}$$

where,

- i, j are respectively the row and the column numbers,
- e_{ij} is the dependency of component c_i to component c_j. We consider e_{ij} as the directed SCA wire from c_i to c_j and E the set of all wires,
- d_{ij} represents the complexity measure of the element e_{ij}, described as follows:
  - if $i = j$, then d_{ij} represents the measure value $d(c_i)$ of component c_i. In our experimentation, we considered by default $d(c_i) = 1$;
  - if $i \neq j$ then d_{ij} represents the number of e_{ij} in E;

2. Second, we calculate the Influence Degree $ID(c_j)$ of component c_j to the whole system. It represents the sum of all elements on column j in $A\_Matrix_{n\times n}$. This is defined as:

$$ID(c_j) = \sum_{i=1}^{n} d_{ij} \tag{2}$$

3. Third, the absolute complexity AC represents the sum of the influence degree of all components. This is defined as:

$$AC = \sum_{i=1}^{n} ID(c_i) \tag{3}$$

4.2 Complexity Measurement Results

The obtained results are presented in Table 2. Column 2 in this table shows the worst architecture complexity values (AC_w) that are obtained by a static code analysis of each candidate system. As we can see, the static architectures of the two candidate systems are very complex and this is particularly true for the largest application. In column 4, we present the number of actions on the graphical user interface in order to describe quantitatively each use case. Columns 5 and 6 present the obtained complexity values and metrics for the recovered use case architectures (UC0 to UC5). We show in column 7 the number of components that are concretely involved in each of the executed use cases.

"Use Case" Architecture Complexity: 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 in column 2). This confirms our intuition that focusing on the runtime "use case"-specific architectures greatly reduces the complexity of the architecture compared to the static one.

Second, column 6 in the table presents the calculated complexity metric (CM) values for all the use case architectures. These values are good for all the recovered use case architectures. However, we noticed that these values decrease when we increase the size of the system. For instance, if we take UCs 3 in the two systems, 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). Besides this, we have executed the same UC (UC 1) on the two systems (which is

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	22	**1659**	**0.29**	**230**
		UC 3	28	**1777**	**0.31**	**242**
		UC 4	35	**1907**	**0.33**	**248**
		UC 5	55	**1941**	**0.34**	**259**
2	**9014**	UC 0	0	**2153**	**0.23**	**392**
		UC 1	4	**2197**	**0.24**	**394**
		UC 2	26	**2398**	**0.26**	**425**
		UC 3	27	**2330**	**0.25**	**413**
		UC 4	30	**2429**	**0.26**	**425**
		UC 5	49	**2885**	**0.32**	**473**

possible, because this UC implies basic functionality in Eclipse), CM in System 2 is less than in System 1 (0.19 vs. 0.21). This is explained by the fact that the complexity (AC_w) of the static architecture increases when we augment the system size, while the complexity values of the use case architectures vary in a stable interval.

Third, we can observe in column 7 that the average number of newly activated components (number of components in the core architecture minus number of components in the UCi architecture) 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 case architectures), which is considered as the initial overhead of our approach. After that, they can focus only on the newly activated components for a specific use case.

By analyzing the recovered use case architectures, we have observed that, in addition to the components of the core architecture there are 44 additional components, which are also common in UC2 to UC5 of System 1, and 9 additional components are common in UC2 to UC5 of System 2. This is explained by the fact that these use cases start with the same actions (creating or importing projects), while UC 1 of the two systems are different (installing a software and file searching).

The last observation that we can make on these results is the high correlation between the number of GUI actions and CM values (correlation coefficient equal to 0.86 for the first system and 0.88 for the second). Though there are only a few measures that are made (5 UCs for each system), there is a general quasi-linear tendency. The more the actions we do on the GUI (the more the UC is complex), the greater CM values we obtain. But CM values remain very low, AC is thereby kept far below AC_w. We experimented a full day working (about 7 hours) on different kinds of projects (12 in total), CM value at the end was equal to 0.54

(the architecture specific to this large UC was almost half less complex than the static one, which is quite a good score).

"Use Case" Service Oriented Architecture Complexity: In the same way, we have evaluated all the spotlighted "use case"-specific Service Oriented Architectures (SOAs). The obtained results are presented in Table 3. This table shows the complexity and number of components in each use case SOA for the two candidate systems, as well as the total number of registered/consumed services by these components. As we can see, the complexity and the number of components is greatly reduced compared to the previous architectures. As consequence, we have eliminated all the noise in these architectures by providing simplified SOA views which may be very helpful for developers who are seeking to make changes only to services.

Table 3: Experiment Results for UC SOAs

	UC SOAs of System 1						UC SOAs of System 2					
	UC0	UC1	UC2	UC3	UC4	UC5	UC0	UC1	UC2	UC3	UC4	UC5
AC	**293**	**295**	**299**	**307**	**305**	**307**	**328**	**328**	**340**	**330**	**332**	**340**
#of Components	**99**	**100**	**102**	**106**	**105**	**106**	**119**	**119**	**125**	**120**	**121**	**126**
# of Registered Services	**86**	**87**	**89**	**90**	**92**	**92**	**89**	**89**	**94**	**89**	**90**	**93**
# of Consumed Services	**305**	**309**	**308**	**309**	**311**	**312**	**321**	**321**	**329**	**322**	**323**	**326**

4.3 Performance Measurement

In order to evaluate the performance of our approach, we have estimated the time for recovering each architecture. Indeed, to run our experiments, we have used 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 of recovering "use case"- specific architectures using our approach.

4.4 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 use case architectures with "ground-truth" use case architectures which correspond to the chosen use cases. A "ground-truth" architecture is the architecture of a software system that has been verified as accurate by the system's architects [11]. Obtaining the "ground-truth" use case architectures is challenging. To mitigate this threat, we have verified manually the component dependencies and the provided/required interfaces (and services) of large parts of the recovered use case architectures by analyzing and checking source code and the meta-data files of the candidate components.

External validity We have implemented and evaluated our recovery approach on set of OSGi based systems which limits our study's generalizability to other kind of systems. To mitigate this threat, we selected systems providing different functionalities (ArchStudio, BPMN2 Designer, BPEL project,...) and sizes.

5 Related Work

A framework comprising a set of principles and processes for recovering systems' ground-truth architectures has been proposed in [11]. The authors of this work have used a set of eight architectures that have been recovered from open source systems and verified as ground-truth architectures by performing a comparative analysis of six software architecture recovery techniques. The authors in [16] have updated these ground-truth architectures to newer versions in order to perform the evaluation. Their results showed that *symbol* dependencies generally produce architectures with higher accuracies than *include* dependencies. Their evaluation showed also that the overall accuracy is low for all recovery techniques. In our work, in order to obtain accurate architectures, we base our recovery process not only on static analysis, but also on dynamic information obtained from execution traces. The dynamic information is then used to enrich the obtained static dependencies, and so enhancing the accuracy.

Most of existing software architecture recovery techniques are based on hierarchical clustering, which seeks to build a hierarchy of clusters starting from implementation level entities. The authors in [18] provide a review of hierarchical clustering techniques in the context of software architecture recovery and modularization. They assert that to employ clustering meaningfully, it is important to understand the particularities of the software domain, as well as the behavior of clustering measures and algorithms in this domain. They provide also an analysis of the behavior of various similarity and distance measures that may be employed for software clustering. In our work, we focus on runtime use case architectures, instead of recovering whole static architectures. However, if the recovered use case architectures using our approach remain complex for a human analysis, the use of one of the existing clustring methods for abstracting those architectures may be helpful to the architect in this case. We organized in three different categories the works that are the closest to what we have proposed in this paper.

5.1 Component-Based Architecture Recovery

In the last two decades there were several works which proposed approaches that aim to recover component-based architectures from different kinds of systems. For instance, the works in [6, 23, 5, 4] focused on extracting component-based architectures from existing object-oriented systems. The works in [6] and [23] are based on the definition of a correspondence model between the code elements and the architectural concepts. In [4] a component is considered as a group of classes collaborating to provide a system function. The interfaces provided and required by a component are the method definitions and calls respectively from and to classes belonging to other components. The identification of components and their interfaces is based on the analysis of traces which are obtained by executing scenarios corresponding to the system use cases. The same authors in [3] proposed the implementation of the recovered architecture in the OSGi framework. Seriai et al. in [22] used Formal Concept Analysis to perform the component interface identification.

Like these works, in our approach we recover architectures of large systems. However, we start from a code that is already based on components. Unlike the works described above, our goal is to recover an architecture of a large-sized system, in which: i) we consider some specific use cases in order to focus on a particular use context and reduce the size of the recovered architecture, and ii) we include dynamic service-oriented features in this architecture

5.2 Service-Oriented Architecture Recovery

Several Service-Oriented Architecture recovery approaches have been proposed in the literature as part of the process of migrating systems to SOA solutions [20]. Most of these approaches are based on static code analysis of the target system. The aim of these approaches is recovering the abstractions and eliciting the legacy fragments that are suitable for migration to SOA. For example, the authors in [19] showed how architecture reconstruction is used as a decision-making tool in the SOA migration process. They enable organizations to understand legacy applications and identify what components can be migrated. The work in [2] proposes to recover UML activity diagrams from legacy system, which are then transformed into BPMN behavior models representing graphically the interactions and collaborations among participants. The authors in [14] proposed to recover behavioral models starting from Web service oriented system, which is a result of migrating existing Web-based applications to SOA solutions. The recovered BPMN models are used to understand the behavior of the new service-oriented application.

Besides, a number of works such as [8, 24, 15] have been proposed to detect SOA patterns [10] from service oriented applications. The authors of [24] proposed to identify service composition patterns by analyzing the execution logs. Demange et al. in [8] have proposed to detect five newly defined SOA patterns specified using a set of rules that combine various static and dynamic metrics. The similarity property between services to detect SOA patterns is used in [15].

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.

5.3 Reducing the Complexity of Architectures

Managing and studying 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 [21]. This matrix is an adjacency matrix which represent module dependencies in a software system. These dependencies are extracted by a conventional static code analysis. The matrix is transformed by eliminating cycles and forming subsystems. In this transformation modules are grouped into composites based on their mutual dependencies. Furthermore, MacCormack et al. in [17] calculate metrics from a DSM in order to measure the degree of modularity of in the architectures of Mozilla and Linux. The authors in [7] have proposed an architectural slicing and abstraction approach for reducing the model complexity. They used the property to check on the software architecture as a slicing criterion. Abi-Antoun et al. [1] proposed a technique to statically extract a hierarchical runtime architecture from object-oriented code. They also analyzed the conformance of an existing architecture with the code. To achieve hierarchy in an object diagram (runtime architecture), they used annotations that developers add to the code in order to assign each object to a single ownership domain that does not change at runtime.

Our approach do not require any annotations to add in the code by developers and we can apply our approach to code that was not specifically designed for this type of processing. In addition, we deal with architectures at a higher level of granularity (component- and service-based ones) and not low level ones (at object-oriented program level).

6 Conclusion and Future Work

Architecture models are important artifacts for understanding the structure and behavior of a given system during maintenance. This kind of artifacts is unfortunately rarely available and if they exist, they are most of the time not up-to-date and do not reflect the system in-hand. This is where architecture recovery plays an important role and provides a precious help for developers. We noticed however 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 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.

Today, there is a need to help the developer to monitor and evolve her/his system directly via its architecture. 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 addition, we want to make them interactive, by enabling developers to start and stop components, and to publish and consume services just by clicking, dragging and dropping the visualized architecture 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. Anquetil, N., Royer, J.C., Andre, P., Ardourel, G., Hnetynka, P., Poch, T., Petrascu, D., Petrascu, V.: Javacompext: Extracting architectural elements from java source code. In: Proc. of WCRE'09. IEEE (2009)
6. Chardigny, S., Seriai, A., Oussalah, M., Tamzalit, D.: Extraction of component-based architecture from object-oriented systems. In: Proc. of WICSA. IEEE (2008)
7. Colangelo, D., Compare, D., Inverardi, P., Pelliccione, P.: Reducing software architecture models complexity: A slicing and abstraction approach. In: Proc. of FORTE'06. Springer
8. Demange, A., Moha, N., Tremblay, G.: Detection of soa patterns. In: Proc. of the 11th ICSOC'13. Springer (2013)
9. Ducasse, S., Pollet, D.: Software architecture reconstruction: A process-oriented taxonomy. IEEE TSE 35(4), 573–591 (2009)
10. Erl, T.: SOA Design Patterns. Prentice Hall (2009)
11. Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: Proc. of IEEE/ACM ASE (2013)
12. Hall, R., Pauls, K., McCulloch, S., Savage, D.: OSGi in action: Creating modular applications in Java. Manning Publications Co. (2011)
13. Jiao, F., Hu, C., Zhao, C.: A software complexity metric for sca specification. In: Proc. of the CSSE. IEEE (2008)
14. 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)
15. 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)
16. 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)
17. 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)
18. Maqbool, O., Babri, H.: Hierarchical clustering for software architecture recovery. IEEE TSE 33(11) (2007)
19. O'Brien, L., Smith, D., Lewis, G.: Supporting migration to services using software architecture reconstruction. In: Proc. of STEP. IEEE (2005)
20. Razavian, M., Lago, P.: A systematic literature review on soa migration. Journal of Software: Evolution and Process 27(5), 337–372 (2015)
21. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: Proc. of the ACM OOPSLA. ACM (2005)
22. Seriai, A., Sadou, S., Sahraoui, H., Hamza, S.: Deriving component interfaces after a restructuring of a legacy system. In: Proc. of WICSA. IEEE (2014)
23. Seriai, A., Sadou, S., Sahraoui, H.A.: Enactment of components extracted from an object- oriented application. In: Proc. ECSA. Springer (2014)
24. 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)