École Polytechnique de Montréal

Génie Informatique et Génie Logiciel

Research proposal submitted
in partial fulfilment of the requirements
for the degree of Doctor of Philosophy (Ph.D.) in Computer Engineering

Detection of SOA Antipatterns

Research Directors:

Dr. Yann-Gaël Guéhéneuc
Dr. Naouel Moha

Prepared by:

Francis Palma

Members of the Jury:

Dr. Pierre-N. Robillard, President
Dr. Yuhong Yan, Member
Dr. Yann-Gaël Guéhéneuc, Member
Dr. Naouel Moha, Member

Abstract

Service-based systems (SBSs), as any other large and complex systems, are subject to change. Changes can be functional (i.e., user requirements) or non-functional (i.e., network protocol, organizations' data security and business policy, and service-level agreements), can be at different level (i.e., design or implementation) and take place during the software evolution and maintenance phases. These changes, often implemented under time constraints, may cause degradation of design and Quality of Service (QoS) when they involve poor solutions, commonly known as *antipatterns*, in opposition to *design patterns*, which are good solutions to recurring problems. Antipatterns in SBSs are commonly referred to as Service Oriented Architecture (SOA) antipatterns. On the one hand, the use of SOA design principles for developing SBSs is emerging. On the other hand, presence of SOA antipatterns in SBSs may (i) result in poor design and QoS and (ii) impact future maintenance and evolution. Therefore, detecting Service Oriented Architecture (SOA) antipatterns deserves attention for assessing the design and QoS of SBSs. Also, this detection may facilitate the future evolution and maintenance of an SBS. Yet, despite of their importance, there are no methods and techniques for specifying and detecting SOA antipatterns. Therefore, we propose a novel approach for detecting SOA antipatterns in SBSs, supported by a framework for specifying and detecting SOA antipatterns. The intended contributions of this work are: (1) an approach to specify and detect SOA antipatterns, (2) a study showing the impact of SOA antipatterns in SBSs, (3) a framework supporting specification and detection of SOA antipattern, (4) a complete tool support for specifying and detecting antipatterns in SBSs, and finally, (5) an empirical validation showing concrete examples of SOA antipatterns detected with our approach.

Keywords: SOA Antipatterns; Service Based Systems; Specification; Detection; Quality of Service; Design; Software Evolution and Maintenance.

Contents

List of Figures

List of Tables

List of Abbreviations

AOL	Abstract Object Language
API	Application Programming Interface
APL	Anti-Pattern Language
AST	Abstract Syntax Tree
BBNs	Bayesian Belief Networks
BNF	Backus–Naur Form
CBSs	Component Based Systems
DSL	Domain Specific Language
FCA	Formal Concept Analysis
GQM	Goal Question Metric
HTTP	Hypertext Transfer Protocol
KDD	Knowledge Discovery in Databases
MDE	Model Driven Engineering
NFR	Non-functional Requirement
NOA	Number of Attributes
NOM	Number of Methods
OO	Object Oriented
QoS	Quality of Service
REST	REpresentational State Transfer
SBSs	Service Based Systems
SCA	Service Component Architecture
SCDL	Service Component Definition Language
SLA	Service-Level Agreement
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SODA	Service Oriented Detection for Antipatterns
SOFA	Service Oriented Framework for Antipatterns
URI	Universal Resource Identifier
WSDL	Web Service Description Language

Preface

This PhD is in cooperation between the École Polytechnique de Montréal under the supervision of Dr. Yann-Gaël Guéhéneuc and the Université du Québec à Montréal under the supervision of Dr. Naouel Moha.

As part of this cooperation, the PhD student is jointly working in the Ptidej Team (Pattern Trace Identification, Detection, and Enhancement in Java) and Latece (Laboratory for Research on Technology for eCommerce) within those respective universities.

Chapter 1

Introduction

1.1 Background

Over the last few decades, software systems have had an exponential growth, while traditional architectural models have already reached at the peak of their capabilities. Moreover, designers and developers must respond quickly to new business requirements and continually reduce the cost more than ever. IBM states that service-oriented architecture (SOA) – a collection of principles and methodologies for designing and developing service-based systems (SBSs), is being promoted as the next evolutionary step to help IT organizations to meet up their business needs [15]. Among other technologies to implement SOA (Section 2.2), *Web services* are the most popular and widely used within industries. Therefore, Web services [3, 5, 52] are more likely to be adopted as the *de facto* standard to deliver effective and reliable consumer transactions and interactions.

In its simplest form, a *Web service* is a way in which two computers or clients communicate with each other over the Internet. SOA can be both an architecture and a programming model, due to its several proven benefits [57], i.e., faster time-to-market, reduced cost, high risk mitigation capability, enhanced security, as also acknowledged by IBM [15]. Therefore, SOA as a current and future programming model is ever demanding, and the use of SOA as an architectural choice is ever increasing too. The wide acceptance of SOA is mainly due to its flexibility and scalability [27]. Service-based systems (SBSs) developed with such architectural style are composed of loosely-coupled and platform independent reusable units, i.e., *Services*, that fulfill some user requirements or provide functional supports.

McKinsey & Company conducted a survey in April 2008 showing that *Software-as-a-Service* (SaaS) and *Service Oriented Architecture* (SOA) are the two most emerging trends for software clients to consume and software engineers to implement. In the survey, about thirty-five percent (out of the 850 surveyed service providers) believe that SaaS would be the most important and practical trend (30 percent in 2006) and 25 percent believe that Web services or SOA is going to be most important and popular (24 percent in 2006) for software development. More significantly, 71 percent of the surveyed subjects were likely willing to adopt SaaS platforms [35]. Norton from Gartner says, "By 2010, 15 percent of large companies will start projects replacing their ERP backbone with a SaaS offering, and 30 to 40 percent of vendors will be offering SaaS service by 2012" [35]. Amazon and eBay are two examples of SBSs.

Yet, the emergence of SBSs cannot overcome common software engineering challenges, e.g., evolution, to fit new user requirements or changes in execution contexts, for example, change of network protocol, change in organizations' data security and business policy or even change in service-level agreements (SLAs). All these changes may degrade the quality of design and quality of service (QoS) of SBSs and may cause the presence of common bad practiced solutions, called *antipatterns*, as opposed to *design patterns*, which are good solutions to recurring problems.

Multi Service and *Tiny Service* are two common and recurring antipatterns in SBSs and it has been shown, in particular, that *Tiny Service* is the root cause of many SOA failures [49]. *Multi Service* is an SOA antipattern that corresponds to a service implementing a multitude of methods related to different business and technical abstractions. Such a service has low reusability and is often unavailable to the end-users [25]. Conversely, *Tiny Service* is a small service with just a few methods, which only implements part of an abstraction. Such service often requires several coupled services to be used together, resulting in higher development complexity and reduced usability [25].

1.1.1 Definitions of Service-Oriented Architecture

Service-oriented architecture (SOA) represents an abstract model of system architecture that utilizes business or application-specific software services, commonly treated as *black-boxes*. Papazoglou *et al.* [67] define *services* as "autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in different ways".

Services in SOA are highly reusable as they are independent and self-contained entities that do not depend on the state of others [40]. Services are typically composed into business processes presented in terms of business concepts regardless of the implementation details. Such processes may be designed by the analysts with the help of CASE tools and can be transformed into executable scripts. Interoperability is also a great technological aspect of SOA. Services in SOA are typically implemented as platform-independent Web services that communicate via XML-based SOAP (Simple Object Access Protocol) protocols and are described using WSDL (Web Service Definition Language) interfaces [37], supporting interoperability between different platforms and programming languages.

Yet, the implementation of Web Services is not limited to the conventional SOAP protocols but another emerging SOA technology REST (REpresentational State Transfer) on top of HTTP (Hypertext Transfer Protocol) is increasingly being adapted within industries [70]. REST is the generic architectural style for modeling web-based applications and resources. In REST, services are defined in a resource-oriented fashion while traditional services are defined more in a function-oriented fashion instead. This manner of modeling service-based systems as a collection of RESTful services is simpler than traditional SOAP-based Web services [70]. This is mainly due to the fact that REST services have a common and standard way of declaring the methods of the interface unlike the SOAP services. To conduct worldwide business and other enterprise-level collaborations in a uniform and interoperable manner, as well as to reuse the existing business resources, SOA relies on Web services and various WS-specifications [81].

1.1.2 Conceptual and Architectural Views of Service-Oriented Architecture

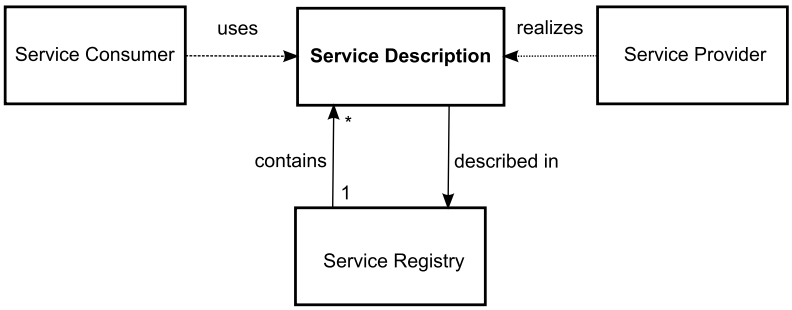

Figure 1.1: Conceptual Model of an SOA

At the conceptual level, service-oriented architecture has three main components: (i) a service provider that realizes and publishes services; (ii) a service consumer that discovers and invokes services; and (iii)

a service registry that maintains a repository of services [7]. This high-level conceptual model is shown in Figure 1.1. From the architectural point of view, SOA contains: (i) the services as "black-box"; (ii) corresponding business processes; and (iii) different integration and management aspects [7]. The overall architectural view of SOA is shown in Figure 1.2 with different layers.

1.1.3 Layers in SOA Reference Model

Here we briefly introduce different layers of SOA reference model shown in Figure 1.2.

Overview of Different Layers in SOA Reference Model

Operational layer includes applications, i.e., J2EE and Microsoft .NET applications supporting business activities, including legacy systems, transaction processing systems and databases [7]. *Service*

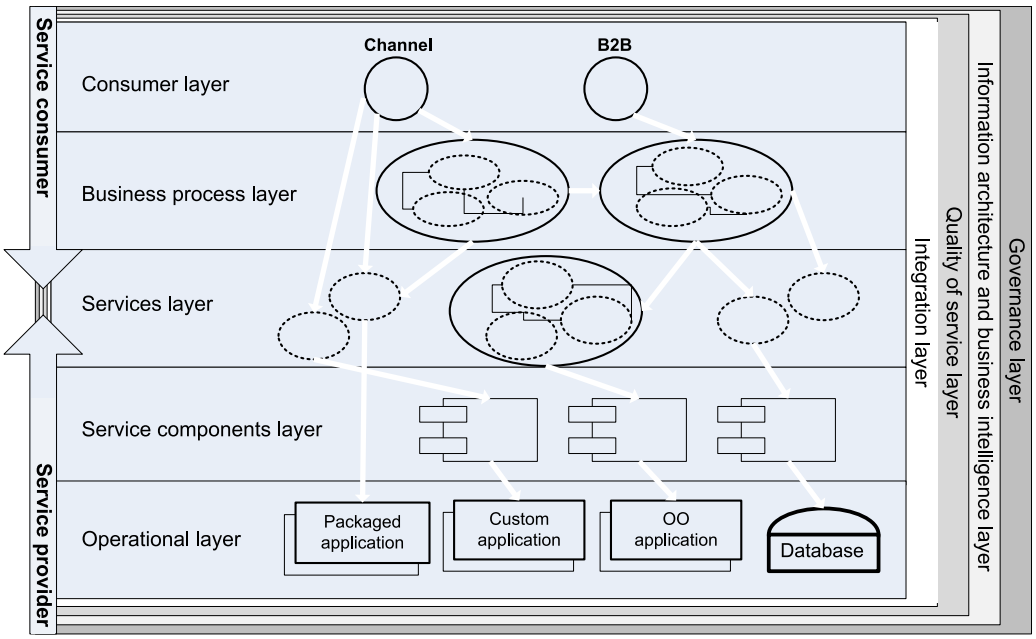

Figure 1.2: An SOA Reference Architectural View [7]

component layer contains software components implementing a service.
Services layer consists of all services defined in an enterprise SOA. All services exposed to the external consumers reside in this layer, from where services can be discovered and invoked or even orchestrated to build a composite service [7].
Business process layer defines the exposed services in the *Services layer*. Service composition is a way to combine groups of services into flows to create applications out of services.
Consumer layer is also known as presentation layer. It provides the means to deliver functions and data to the end users to meet a particular requirement.
Integration layer enables the mediating, routing, and transporting service requests from the requester to the provider [7].
QoS layer ensures an SOA fulfilling nonfunctional requirements (NFRs). To ensure NFRs, the QoS layer captures, monitors, logs, and accordingly signals deviations [7].
Information architecture and business intelligence layer captures inter-industry and industry-specific data structures and business protocols for exchanging business data among industries and partners [7].
Governance layer covers all aspects of business operational life-cycle management in SOA. It offers

guidance and policies for making decisions about an SOA and supervising all aspects of an SOA solution, including capacity, performance, security, and monitoring [7].

1.1.4 Different SOA Technologies

Different SOA implementation technologies [16, 37, 70, 74, 81] (described in Section 2.2) are already discussed in the literature. Those different SOA implementation technologies are important for this research, hence require an insight. Detection techniques for different SBSs designed and developed with different SOA technologies might vary. Hence, we might have different strategies for detecting SOA antipatterns in SBSs developed with different SOA technologies. Therefore, we include Section 2.2 (optional for the readers) to introduce those SOA technologies in brief.

Service-based systems are developed on top of SOA principles. For implementing SOA, a number of different technologies are available. REST-style [28] or SCA [16] are examples of popular SOA implementation technologies. In particular, we briefly describe WSDL-based Web Services (Section 2.2.1), SOAP/RPC-based Web Services (Section 2.2.2), Service Component Architecture (Section 2.2.3), and REST-style (Section 2.2.4).

1.1.5 Outline

The remainder of this proposal is organized as follows. The next few sections discuss the problem to be solved including the motivation. Chapter 2 summarizes the related work. We present the thesis, challenges, and our proposed solution in Chapter 3. Chapter 4 introduces the proposed approach, the framework, and the preliminary experiments along with the results. Chapter 5 presents the current state of the research, and highlights our key current contributions and future research directions. Finally, Chapter 6 concludes with a possible publication schedule.

1.2 Motivation

Assessing design and QoS in SBSs helps to ease maintenance and evolution of SBSs. Maintainability is the quality factor that includes all the features of a software to make it easier to maintain [32]. Among all the development phases, the maintenance phase lasts longer and it requires more expenses and resources with time [38]. With the maturity of a system, software companies intend to maximize productivity leaving the maintenance issues as a secondary concern. Software maintenance is the process of modifying through updating or repairing software, ensuring its initial functionalities unchanged [11]. It is shown that maintenance and evolution phase consume resources of more than 60% [39, 66]. Also, after an extensive survey on "traditional" softwares, Lientz [53] discovered that 70% of all cost of a software product is spent on its maintenance. In an empirical study, Tan and Mookerjee [79] showed that the more time spent or delayed to perform maintenance tasks, the more the cost rises due to design degradation with the system maturity. Belady and Lehman [8] define software evolution as the dynamic behavior of software systems as they are maintained and enhanced over times. Either the evolution of a software may introduce new antipatterns or the existing antipatterns might hinder software evolution process.

As the usage of SOA architecture and its different underlying technologies to implement SBSs is emerging; the existing diverse technologies (Section 2.2) must also be considered, while analyzing and assessing the SBSs, i.e., considering SBSs developed with diverse technologies. Assessing design corresponds to the static analysis of an SBS while assessing the QoS corresponds to the measurement of dynamic aspects (i.e., performance, security, transactions etc.) of an SBS. One of the objectives of analyzing and assessing the design and QoS criteria of SBSs is to detect antipatterns in them. The detection of the SOA antipatterns within SBSs is also a part of maintenance activities and it is important to detect them at the

earliest possible. In summary, by proposing an approach for specifying and detecting SOA antipatterns, we can contribute to better maintenance and evolution by assessing design and QoS of SBSs.

1.2.1 Importance of Antipatterns

Antipatterns are *bad* solutions to *recurring* design problems. As stated by Brown *et al.* [13], antipatterns describe solutions to the commonly occurring problems but generate negative consequences on the quality of systems. Therefore, when a design creates more problems than it solves, it becomes an antipattern.

A software project can be designed, developed, and managed more effectively by having insight into the causes, symptoms, and consequences of the negative solutions. Even after the project is finished, antipatterns can be detected and can be refactored, i.e., by changing the design and implementation, and, successful refactored solutions may make the antipatterns beneficial.

We assume that SOA antipatterns have the impact on design and QoS of SBSs, and affect the maintenance and evolution of any SBSs. This is also important to validate the fact that the SOA antipatterns are harmful for any target systems, in particular for SBSs.

1.2.2 Impacts of Antipatterns

Antipatterns may have the impact on the understandability and analyzability (hence, on the maintainability), and also on the changeability (hence, on the evolvability) of a program. Understandability of a program is the ease to which the program is comprehensible with regard to its goal and design [1]. Smells and antipatterns impact the understandability of the programs. Analyzability and understandability are parts of maintainability according to ISO software quality model [1]. Software analyzability refers to the ease one can analyze a software. Abbes *et al.* [4] established a quantitative evidence on the relations between antipatterns and program comprehension, i.e., understandability of a program. The authors perform experiments to see if programs with the antipatterns are harder to understand than the programs without any antipatterns and show that the presence of multiple antipatterns negatively impact the program comprehension or understandability, and hence, the maintainability of the program. The results also support *a posteriori* the removal of antipatterns as early as possible from systems and, therefore, the importance and usefulness of the detection of antipatterns [4].

Changeability refers to the degree to which a system can be changed easily. Better changeability refers to better evolvability. Antipatterns have the impact on the changeability of the programs as well. Khomh *et al.* [46] also provided empirical evidence, that the entities (i.e., classes) participating in antipatterns are more expected to be changed, and to be involved in faults than other non-participating classes, meaning that antipatterns have the impact on the changeability of systems.

Software *evolvability* refers to the ease of further developing a software. IEEE has defined software maintainability as "the ease to which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment" [2]. Mäntylä and Lassenius also showed with an empirical study that software systems with smells have less evolvability than the systems without smells [59].

All the above evidences are established for Object-oriented (OO) systems. However, these evidences are suitably true (and should be proven) for other paradigms, in particular for SBSs. Hence, to improve the maintainability and evolvability, and to assess the design and QoS of SBSs, detection of SOA antipatterns in SBSs is important and needs research attention.

Chapter 2

Related Work

For SBSs, quality design is essential for building easily maintainable, and evolvable systems. Patterns and antipatterns are two best ways to express such quality concerns and there is a common belief that applying patterns improves software quality and reusability [33, 34]. Gamma *et al.* first provide the basic concepts on design patterns for Object-oriented (OO) paradigm. Brown *et al.* describe an antipattern as "literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences", and introduce a collection of 40 antipatterns. However, methods and techniques for the specification of SOA antipatterns are still in their infancy in compare to OO patterns and antipatterns.

In contrast with OO antipatterns, there are very few books and papers dealing with SOA antipatterns. Most references are Web sites where SOA practitioners and SOA antipattern experts share their experiences in SOA design, development, and antipattern [18, 42, 63]. For example, Cherbakov *et al.* [18] describe 9 SOA adoption, design and realization related antipatterns. The authors identify the problem and its root cause, and provide some symptoms with possible solutions for each SOA antipattern, but do not provide any concrete evidence to specify or detect them. Jones [42] also describe 13 SOA antipatterns and their resolutions which may help for correction but not for specification or detection. Modi [63] enlisted 7 new SOA antipatterns with the very high level overview. Dudney *et al.* [25] published the first book on SOA antipatterns. This book provides a catalog of approximately 50 antipatterns related to the architecture, design and implementation of systems based on J2EE technologies, such as EJB, JSP, Servlet, and Web services. However, the authors did not put any light on the detection of these antipatterns. Another book by Rotem-Gal-Oz *et al.* [73] is published recently that describes some new SOA patterns and antipatterns. Král *et al.* [49] described seven antipatterns, which are caused by an improper use of SOA standards in their recent paper.

Application and recovery of design pattern may help with better program comprehension and maintainability, likewise, detection of antipatterns may help in improving programs performance, and better evolution and maintenance in general. There are numerous approaches [12, 23, 29, 45, 51, 55, 60, 61, 64, 68, 69, 75–78, 82] that deal with the detection of OO code smells, antipatterns or even design patterns. However, unlike the research on detecting OO issues, research on revealing methods and techniques for the detection of SOA antipatterns is in their infancy. In Section 2.1, we discuss briefly some approaches proposed in the literature to detect OO smells and antipatterns. Section 2.2 highlights some common service-oriented implementation technologies, and in Section 2.3 we summarize the contributions on the detection of SOA antipatterns from the literature.

2.1 Detection of OO Smells and Antipatterns

Numerous works have been done to detect and locate smells in the OO systems. Those proposed approaches in the literature are based on different techniques, i.e., graph solving [47], machine learning [58], rules-based [64], and probability matrix [71].

Among the above works, the DECOR [64], a rule based method for the specification and detection of code and design smells by Moha *et al.* is closely related to our work. The authors suggest to use a unified vocabulary and a domain specific language to manually specify smells, and then automatically generate smell detection algorithms which are directly executable. Despite of good detection performance by DECOR [64], i.e., precision of 60.5% and recall of 100%, the main shortcoming of the method relies in its technology agnosticism feature, meaning that it can only perform detection for OO systems. Indeed, DECOR was thoroughly validated only with one OO system (Xerces v2.7.0). Validation with more OO systems could strengthen the approach, i.e., increase the precision. Being inspired from DECOR, we intend to follow slightly similar approach in the context of service orientation for detecting SOA antipattern.

Also for detecting design defects or antipatterns in OO systems, a number of approaches were proposed in the literature. Among them, metrics-based [12, 29, 60, 61, 75], combining metrics-based and machine learning technique [51], ontology based [55, 76, 77] and rule-based [23, 45, 69, 78] approaches are worth to mention.

Marinescu [61] introduces a rule card-based approach to detect design smells within OO systems. For the detection, the author performs a quantitative analysis of the design smells and identifies the relationships among them to define *detection strategies*, stated as a grouping of metrics for applying to the target system to detect design smells. Using his proposed approach, Marinescu detects nine common design smells in a C++ application. However, the proposed approach is not applicable to Java systems. Furthermore, the approach is validated only for one case study. Our approach may also use this useful rule-based technique for detecting SOA antipatterns with an improvement in understandability of rule cards, meaning that our rule cards will be in high level language, whereas, Marinescu used SOD [19], a machine understandable language.

Using similar metric-based approach combining with a machine learning technique, Kreimer [51] proposes an adaptive method towards the detection of design smells. Kreimer allocates a set of OO metrics to each design smell and computes those metrics with only static program analysis for producing a program dependency graph (i.e., an abstract model). From that model, values are fed to a machine learning technique, that is already trained with example design smells. To this end, decision trees are used for the final classification. The detection performance by the constructed decision tree depends on the training set, and very often the training set is not instantly available. Even, the training phase is not automatic, i.e., program locations having possible design smells are chosen manually. In the detection phase, user decides which part of the system needs to be analyzed. Finally, user has to verify all suspicious cases, and, approves or rejects the suggestion which requires good amount of human involvement. More importantly, the approach needs more training data for better accuracy. In contrast to Kreimer's approach, we perform also dynamic analyses and try to minimize human intervention as much as possible, i.e., generating the detection algorithms automatically.

Parsons and Murphy [68] propose a framework for automatically detecting and assessing the impact of poor performance design, i.e., performance antipatterns, in component-based systems (CBSs) using run time analyses. But their approach performs detection only in component-based enterprise systems, in particular J2EE applications, using a rule-based approach relying on static and dynamic analysis. The authors detected J2EE 12 antipatterns introduced by Dudney in his book, *J2EE Antipatterns* [25]. The approach by Parsons and Murphy is similar to us apart from the fact that we focus on SBSs and

underlying all SOA technologies, which is more challenging in terms of antipatterns detection.

Kessentini *et al.* [45] also propose an approach for the detection and correction of design smells in source code. The key feature of the proposed approach is the allowance to automatically find detection rules, where rules are defined as aggregated metrics (similar to the works by Marinescu [61] and Kreimer [51]). The authors use *genetic programming* for rule extraction, i.e., to find a optimal set of rules, where the best set of rules is the one that maximize the smell detection. We may also use different heuristics, i.e., genetic programming for constructing rules to maximize SOA antipattern detection. Kessentini *et al.* perform detection for only three design smells with the precision and recall ranging between 81% and 100% in four OO systems. The authors show that their approach outperforms DECOR [64], a state-of-the-art metric-based approach, where rules are defined manually.

In another work, Wong *et al.* [82] used a genetic algorithm for detecting software faults and anomalous behavior related to the resource usage of a system (e.g., memory usage, processor usage, thread count). Their approach is based on *utility functions*, which correspond to predicates that identify suspicious behavior based on resource usage metrics. For example, a utility function may report an anomalous behavior corresponding to spam sending if it detects a large number of threads. However, this approach by Wong *et al.* is not applicable to design or behavioral antipatterns, and not even to SBSs.

There are also number of works done on the detection of OO design patterns. Among them the approaches based on static and dynamic analyses [41], graph theory [80], rules [50], string matching algorithm [43, 43], and metrics-based [6] are worth to mention.

2.2 Different SOA Technologies

In the following sections, we present different SOA technologies that are being used now-a-days for building Service-based systems (SBSs). Among them WSDL-based Web Services (Section 2.2.1), SOAP/RPC-based Web Services (Section 2.2.2), Service Component Architecture (Section 2.2.3) and REST-style (Section 2.2.4) are noteworthy. It is important for this work detailing their differences from the architectural perspective to show why we might need different strategies for detecting SOA antipatterns in SBSs developed with diverse SOA implementation technologies.

2.2.1 WSDL-based Web Services

The *Service Web* (i.e., the network of services) is built on Internet standards [74]. Internet standards focus on protocols and not on implementations of Services. Three major standards for Web services are: (i) SOAP, a standard for messaging over HTTP and other Internet protocols, (ii) WSDL, a language for precisely describing the programmatic interfaces of Web services, and, (iii) UDDI, a business registry standard for indexing Web services, so that their descriptions can be located by development tools and applications. In general, SOAP, WSDL, and UDDI form the infrastructure for Web services [74]. Web services enable interoperability of applications that use different hardwares, operating systems, or programming languages. Figure 2.1 shows an example of a SOAP request. In Figure 2.1, the request is being sent to the URL http://www.stockquoteserver.com/StockQuote. The request specifies that the *GetLastTradePrice* operation is to be performed using the symbol DIS.

Figure 2.2 shows an example of a SOAP response. In Figure 2.2, the *GetLastTradePriceResponse* element contains the DIS price of 34.5. SOAP is being used in an RPC style in this example. However, SOAP can also be used for one-way message delivery.

Figure 2.3 shows an example of an WSDL definition of a Web service, which describes the *StockQuote* service. From Figure 2.3, we can see it begins with type definitions for *GetLastTradePrice* and *GetLast-*

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://example.com/stockquote.xsd"

<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <SOAP-ENV:Body>
                <m:GetLastTradePrice xmlns:m="http://example.com/stockquote.xsd">
                        <symbol>DIS</symbol>
                </m:GetLastTradePrice>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.1: An Example of SOAP Request

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        <SOAP-ENV:Body>
                <m:GetLastTradePriceResponse xmlns:m="http://example.com/stockquote.xsd">
                        <Price>34.5</Price>
                </m:GetLastTradePriceResponse>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.2: An Example of SOAP Response

TradePriceResponse using XML Schema. The remainder of the Figure 2.3 builds up the Web service definition by assembling the types into messages, the messages into operations, and the operations into port types. Then, the WSDL binds the port type to the SOAP/HTTP protocol, and, finally assigns it the address http://www.stockquoteserver.com.

2.2.2 SOAP/RPC-based Web Services

A remote procedure call (RPC) is an inter-process communication through which a procedure or function can be to executed in another machine regardless of detail coding for such remote interaction [17]. SOAP RPC-based Web services are remote procedure calls to Web services via SOAP, and are easier to run as platform-independent system entities and a good choice for many SOA designers till now. Perhaps, this preference will not be for long time with the rise of REST, SCA and other newer technologies. Figure 2.4 shows a simple model to explain how the clients and Web services interact with SOAP RPC requests and responses.

Figure 2.5 shows the SOAP RPC request to the Web service from a client. According to RPC specification, the request content is within SOAP body, and the name of the element, i.e., *openAccount* matches the name of the method to be called from Web service [17]. Figure 2.6 shows the response to the previous SOAP RPC request.

This conventional SOAP messaging use a message encoding, and the most common one is document-style SOAP [17]. But there are other alternatives to SOAP encoding to make the SOAP message more

```
<?xml version="1.0"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd" xmlns:soap="...wsdl/soap/" xmlns="...wsdl/">
   <types>
      <schema targetNamespace="http://example.com/stockquote.xsd" xmlns="http://www.w3.org/1999/XMLSchema">
         <element name="GetLastTradePrice">
            <complexType>
               <all> <element name="symbol" type="string"/> </all>
            </complexType> </element>
         <element name="GetLastTradePriceResponse">
            <complexType> <all> <element name="Price" type="float"/> </all> </complexType>
         </element> </schema> </types>
   <message name="GetLastTradePriceInput">
      <part name="body" element="xsd1:GetLastTradePrice"/> </message>
   <message name="GetLastTradePriceOutput">
      <part name="body" element="xsd1:GetLastTradePriceResponse"/> </message>
   <portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
         <input message="tns:GetLastTradePriceInput"/>
         <output message="tns:GetLastTradePriceOutput"/> </operation>
   </portType>
   <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="GetLastTradePrice">
         <soap:operation soapAction="http://example.com/stockquote.xsd"/>
         <input>
            <soap:body use="encoded" namespace="http://example.com/stockquote.xsd"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/> </input>
         <output> <soap:body use="encoded" namespace="http://example.com/stockquote.xsd"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/> </output>
      </operation> </binding>
   <service name="StockQuoteService"> <documentation>My first service</documentation>
      <port name="StockQuotePort" binding="tns:StockQuoteBinding">
         <soap:address location="http://www.stockquoteserver.com/StockQuote"/> </port> </service>
</definitions>
```

Figure 2.3: An Example of an WSDL of a Web Service [74]

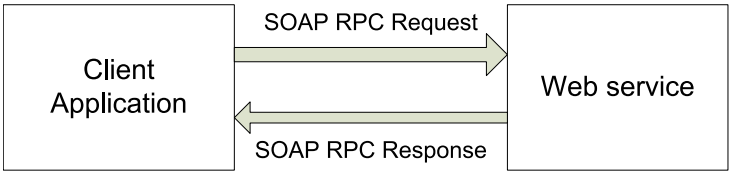

Figure 2.4: Interacting with a Web Service via SOAP RPC

secure by providing the attribute value *encodingStyle* within the SOAP message. In this case, the encoding of header or body content is done according to the rules and constraints of someone's own data model and schema, rather than using default SOAP Data Model. Figure 2.7 shows different ways we can package SOAP messages.

Document – Document-style SOAP refers to the way where the application payload is placed directly within the SOAP Body element as a child [17]. In Figure 2.7, as shown on the left, the application content is a direct child of the <soap:Body> element.

RPC – RPC-style SOAP wraps the application content inside an element whose name can be used to indicate the name of a method to dispatch the content to [17]. In Figure 2.7, as shown on the right, we can see the application content is wrapped using <m:purchase> element.

Literal – Literal SOAP messages use any schemas for providing the meta-level description of the SOAP payload [17]. Therefore, using literal SOAP is similar to taking an instance document of a particular schema and embedding it into the SOAP *Body*, as we can see at the top of Figure 2.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope">
    <env:Body>
        <bank:openAccount env:encodingStyle=
         "http://www.w3.org/2002/06/soap-encoding"
         xmlns:bank="http://bank.example.org/account"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <bank:title xsi:type="xs:string">
                Mr
            </bank:title>
            <bank:surname xsi:type="xs:string">
                Bond
            </bank:surname>
            <bank:firstname xsi:type="xs:string">
                James
            </bank:firstname>
            <bank:postcode xsi:type="xs:string">
                S1 3AZ
            </bank:postcode>
            <bank:telephone xsi:type="xs:string">
                09876 123456
            </bank:telephone>
        </bank:openAccount>
    </env:Body>
</env:Envelope>
```

Figure 2.5: A SOAP RPC Request [17]

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope">
    <env:Body>
        <bank:openAccountResponse env:encodingStyle=
         "http://www.w3.org/2002/06/soap-encoding" xmlns:rpc=
         "http://www.w3.org/2002/06/soap-rpc" xmlns:bank=
         "http://bank.example.org/account" xmlns:xs=
         "http://www.w3.org/2001/XMLSchema" xmlns:xsi=
          "http://www.w3.org/2001/XMLSchema-instance">
            <rpc:result>bank:accountNo</rpc:result>
            <bank:accountNo xsi:type="xsd:int">
                10000014
            </bank:accountNo>
        </bank:openAccountResponse>
    </env:Body>
</env:Envelope>
```

Figure 2.6: A SOAP RPC Response [17]

Encoded – By transforming application-level data structures via the SOAP Data Model into a XML format conforming to the SOAP Schema, one can easily create the SOAP-encoded messages. These encoded messages are machine-produced alike and not look like as the same message expressed as a literal encoding [17]. Figure 2.7 (bottom) shows the Encoded messages used in SOAP Body.

The SOAP specification provides four ways to encode SOAP messages (Figure 2.7). But, Web services mostly use SOAP Encoded-RPC and SOAP Document-Literal [17]. The Document-Encoded and RPC-

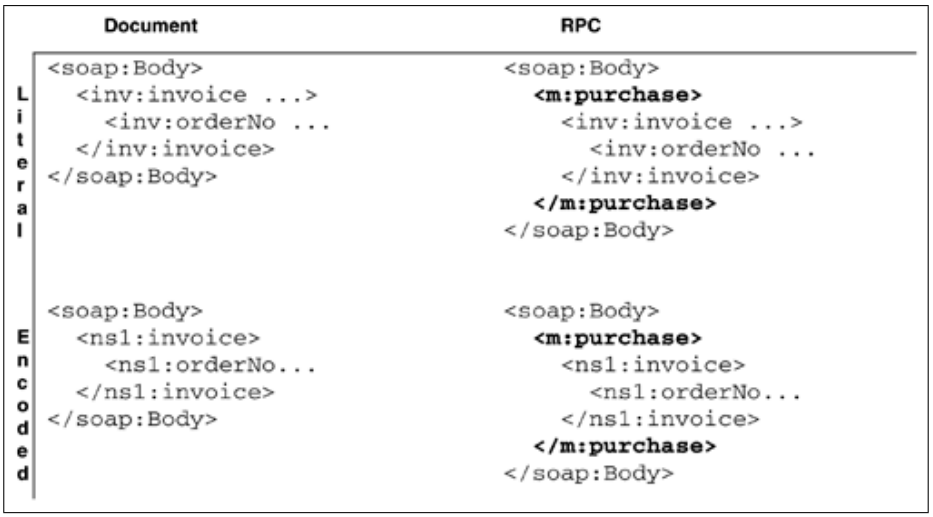

Figure 2.7: Document, RPC, Literal, and Encoded SOAP Messages [17]

Literal are rarely used in practice.

2.2.3 Service Component Architecture

Service-Oriented Architecture (SOA) is a framework combining individual business functions and processes, i.e., services for building SBSs. On top of SOA, Service Component Architecture (SCA) is a relatively newer specification that defines a model for building SBSs. One of the main advantage of SCA is that it provides the developers free from traditional middleware programming by abstracting it from business logic allowing them to focus on business logic. Other advantages of SCA include simplified business component development, simplified assembly and deployment of business solutions; increased portability and reusability; and improved testability [16].

Using SCA to build a service-oriented application has two main steps: (i) the implementation of components, and (ii) the assembly of components to build the application. Some major SCA concepts are briefly discussed as follows:

SCA Components and Composites – All SCA systems are built from one or more components, i.e., Java classes running in a single process or on different machines, those rely on some communication mechanism to interact with each other. And in the SOA perspective, these interactions are modeled as *services*. Figure 2.8 shows an example of an SCA component. On the other hand, a composite is a higher level logical construct with one or more components running in a single process, on a single computer or are distributed across multiple processes on multiple computers. A complete SCA application can be constructed of just one composite, or can combine several composites. An SCA composite is typically described in a configuration file with the extension *.composite* which is an XML document written in Service Component Definition Language (SCDL) to describe the components of the composite and how they relate. Figure 2.9 shows a basic composite model with different parts.

SCA components are technology neutral. Indeed, each component relies on a common set of abstractions, including services, references, properties, and bindings, to specify its interactions with the outer world [16]. Each component usually implements some business logic, exposed as services. A Java component, may describe its services using Java interfaces, whereas a BPEL component may describe its services using the Web Services Description Language (WSDL). A component might also rely on services provided by other components inside or outside of its domain, and to locate

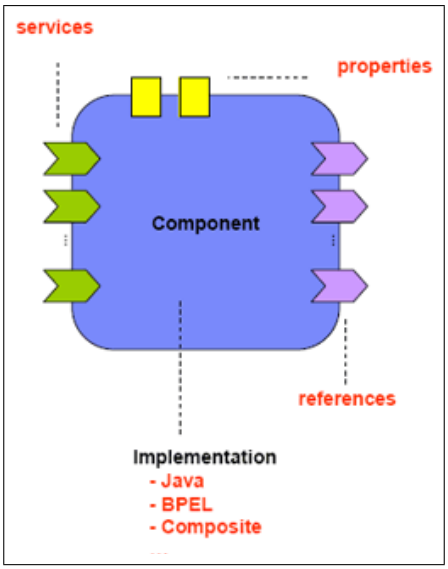

Figure 2.8: An SCA Component Model [62]

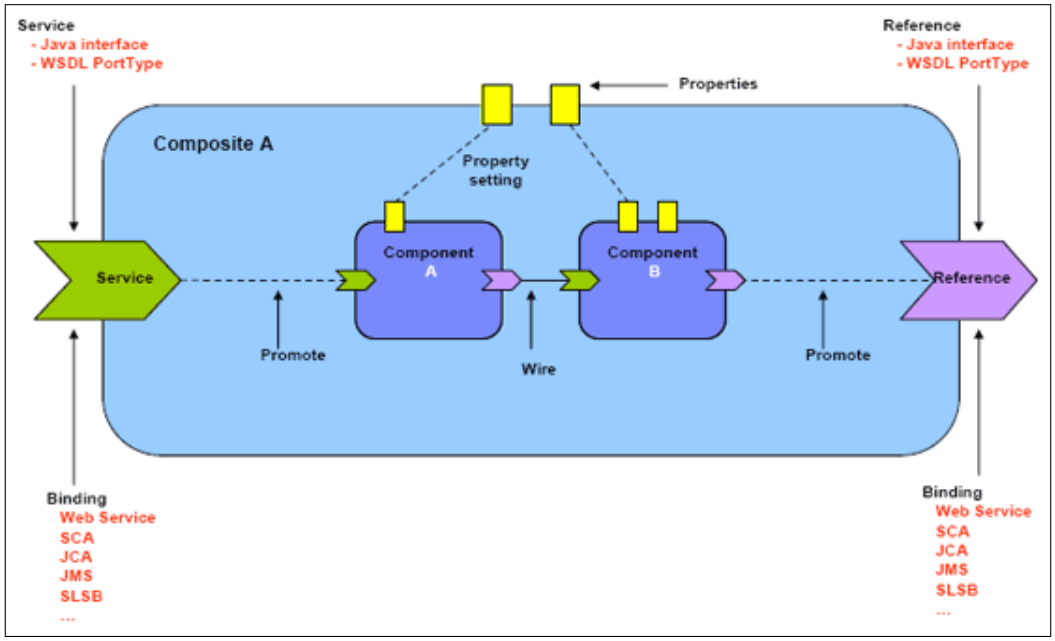

Figure 2.9: An SCA Composite Model [62]

the dependent component or service the component may use references. Along with services and references, a component may also define one or more properties. Each property contains a value that can be read by that component from the SCDL configuration file (i.e., *.composite* file) during its instantiation.

Domains – An SCA application may run in a specific environment installed by a single vendor, that can be a primary example of a domain. A domain may have more than one composite, implemented in several processes, running on several machines. Figure 2.10 shows an example of an SCA domain. Even though an SCA composite runs in a single-vendor environment, it can also communicate with applications in other domain. Here come the Web services, through an interoperable protocol to make themselves accessible from outside. Figure 2.11 shows such two SCA domains each with two computers. To communicate between domains, or with other non-SCA applications, a component will typically allow

access through Web services.

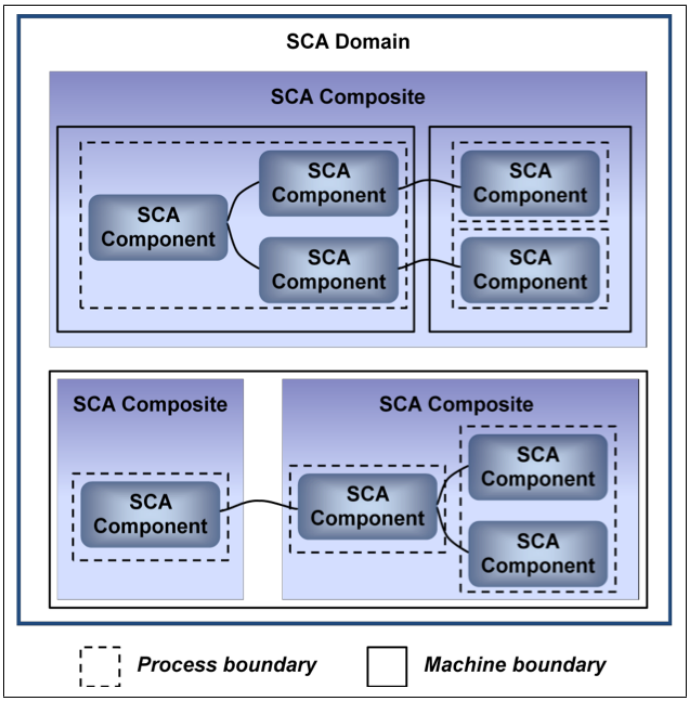

Figure 2.10: An Example of an SCA Domain [62]

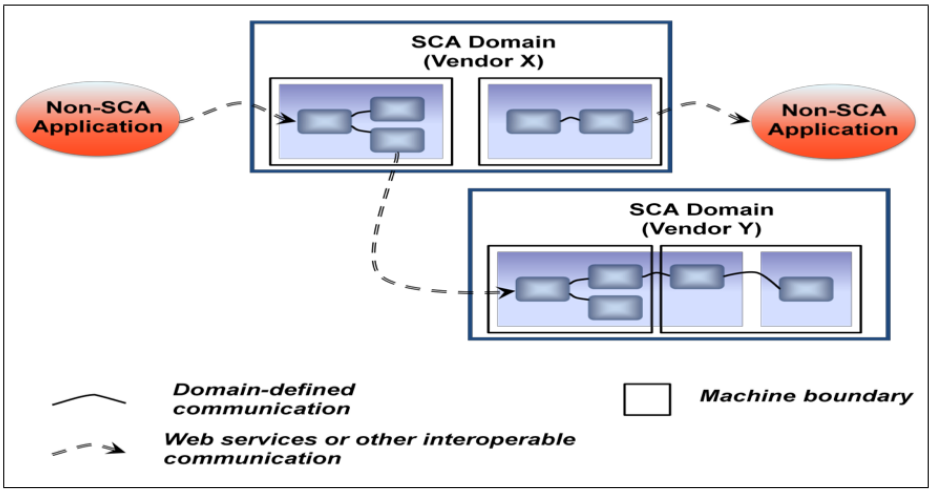

Figure 2.11: An SCA Domain Model: Interaction with Another Domain or non-SCA Application [62]

Bindings – Services and references allow a component to communicate with other systems while the binding specifies how this communication will be done. Although, having explicit bindings are optional for the components. More specifically, a component communicating with another component in the same domain does not need to have any explicit bindings. Rather, the runtime determines the bindings, However, for outer domains, the developer must specify the bindings, which defines a protocol to be used to communicate with this service. One of the main advantage of SCA over the older technologies is that, bindings are separated from the component implementation and business logic, whereas older technologies mix them up. Figure 2.12 shows the binding position in an SCA application. SCA provides an approach for wrapping different protocols into distinct technologies with different application program interfaces

(APIs) through allowing each remote service to specify the protocols it supports using bindings [16]. Figure 2.13 shows different binding types used in SCA.

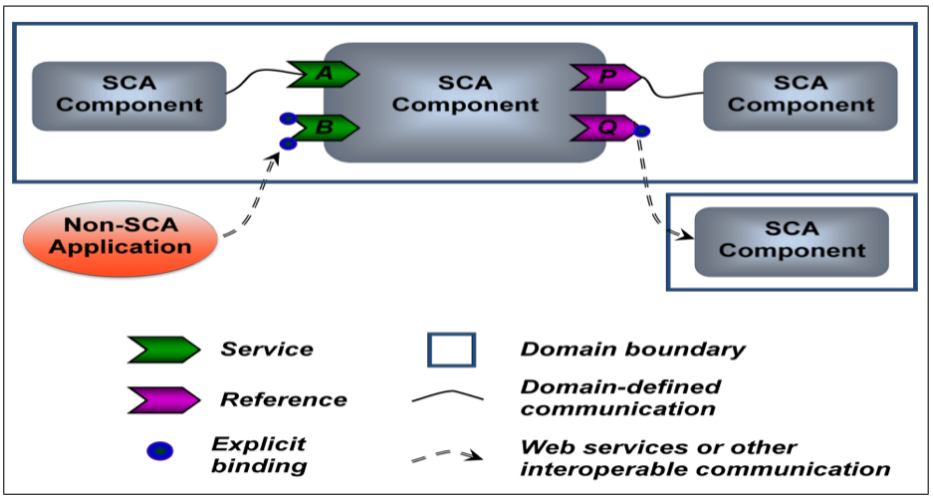

Figure 2.12: The Position of an SCA Bindings [62]

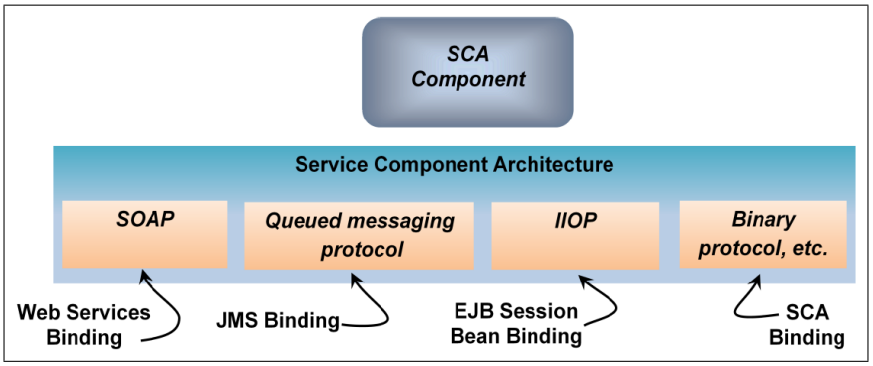

Figure 2.13: Different SCA Bindings [62]

The Architectural View of an SCA Application – Figure 2.14 shows the architectural view of an SCA application. As an alternative to older technologies (i.e., EJB and JAX-WS), SCA introduces a new way to create Java business logic for a service-oriented world by providing the assembly mechanism for components implemented using various technologies, providing a greater interoperability at the end.

2.2.4 REST-style

Fielding first coined the term *REpresentational State Transfer* (REST) in his doctoral dissertation in 2000 [28]. REST describes an architectural style of Web applications and a collection of architectural constraints and principles. An application or design meeting REST constraints and principles is named as RESTful. One of the most important REST principles for Web applications is that the interaction between the client and server must be stateless between requests [28]. On the server side, the application state and functionality are divided into resources. Example of the resource can be an application object, a database record, or an algorithm, and each resource is uniquely addressable using a URI (Universal Resource Identifier). Standard HTTP methods such as GET (for reading data), PUT (for updating data), POST (for creating data), and DELETE (for deleting data) are used in REST.

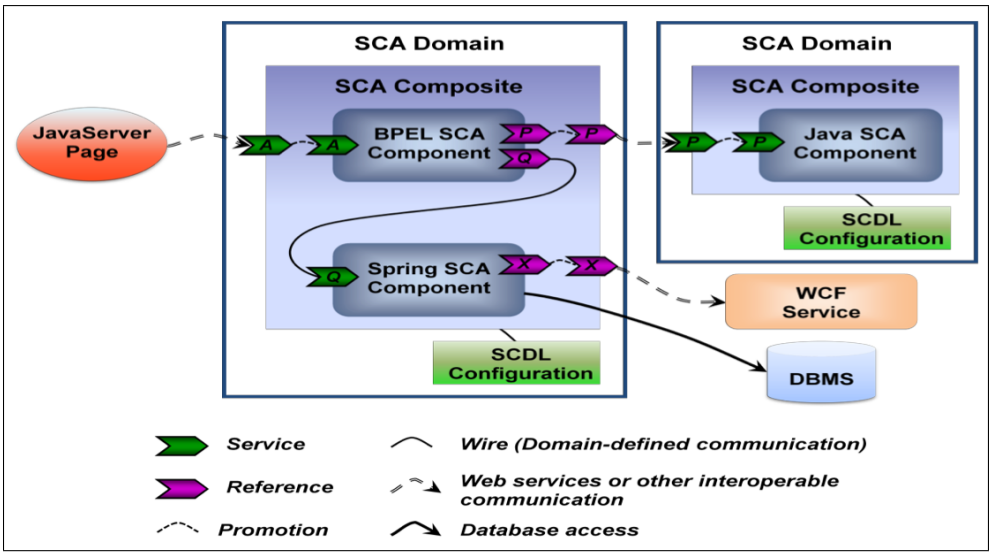

Figure 2.14: The Architectural View of an SCA Application [62]

REST Constraints

Any architecture meeting the following REST constraints can be classified as RESTful [28].
Client–server – A uniform interface separates clients from servers;
Stateless – No client context being stored on the server between requests;
Cacheable – Responses must define themselves as cacheable, eliminating excess client–server interactions and improving scalability and performance;
Layered system – Intermediary servers may improve system scalability
Code on demand – Servers are able to customize the functionality of a client by the transfer of executable code;
Uniform interface – The uniform interface between clients and servers, simplifies the architecture.

From Figure 2.15 and Table 2.1, one can easily notice how simple is the REST compared to traditional SOAP. For example, using Web services and SOAP, a request would look something like in Figure 2.15, whereas, using REST, the same request would look like in Table 2.1. This URL is sent to the server using a simpler GET request, and the HTTP reply is the raw result data that one can use directly.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 <soap:body pb="http://www.acme.com/phonebook">
  <pb:GetUserDetails>
   <pb:UserID>12345</pb:UserID>
  </pb:GetUserDetails>
 </soap:Body>
</soap:Envelope>
```

Figure 2.15: An Example of an SOAP Message

Fielding in his thesis [28], shows step by step how a RESTful system can be built over the current Web

http://www.acme.com/phonebook/UserDetails/12345

Table 2.1: An Example of an REST URI

by adding each constraints which are essential for a system to be RESTful.

Starting with the Null Style – The Null style, in Figure 2.16 is simply an empty set of constraints, from the architectural perspective which describes a system there are no differentiating boundaries between components. For REST it is the starting point to build.

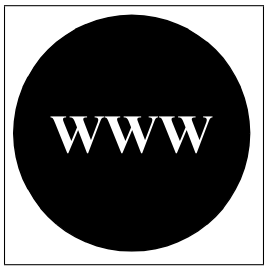

Figure 2.16: Null Style [28]

Client-Server – The first constraint to be added is the client-server architectural style as shown in Figure 2.17, keeping in mind the principal of *separation of concerns* that allows the components to evolve independently.

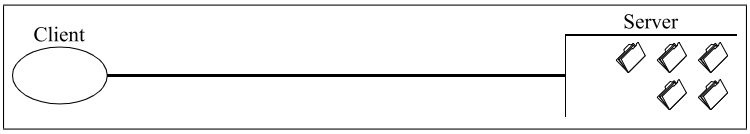

Figure 2.17: Client Server Architectural Style [28]

Stateless – Communication should be stateless in such a way that each request from client to server must contain all of the information necessary to understand the request. This addition of statelessness is shown in Figure 2.18.

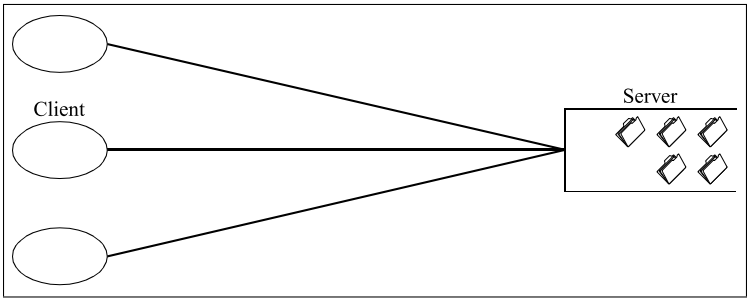

Figure 2.18: Client Stateless Server [28]

Cache – To improve network efficiency, cache constraints can be added to form the client-cache-stateless-server style. A response is cacheable meaning that a client cache can reuse any response data for any

successive equivalent requests. Figure 2.19 shows such addition of client cache.

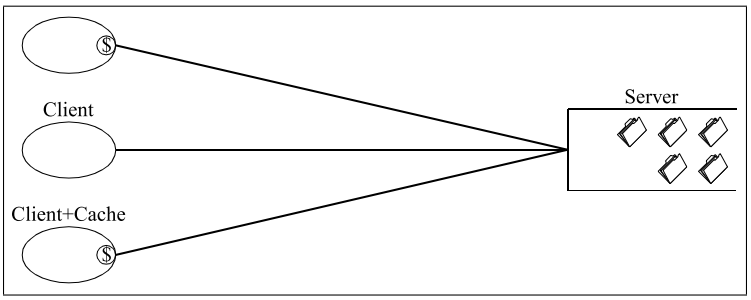

Figure 2.19: Client Cache Stateless Server [28]

Uniform Interface – REST emphasizes on a uniform interface between components as shown in Figure 2.20. By applying the software engineering principle of generality to the component interface, implementations are separated from the services they provide.

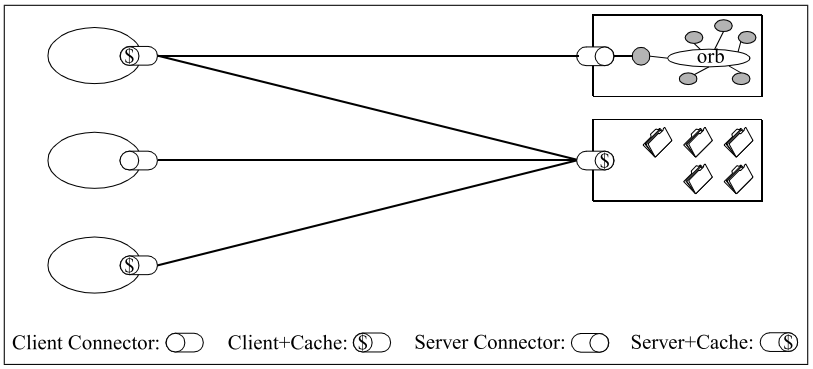

Figure 2.20: Uniform Client Cache Stateless Server [28]

Layered System – Adding layered system constraints improves performance for Internet-scale requirements, as shown in Figure 2.21. Layering allows an architecture to be composed of hierarchical layers.

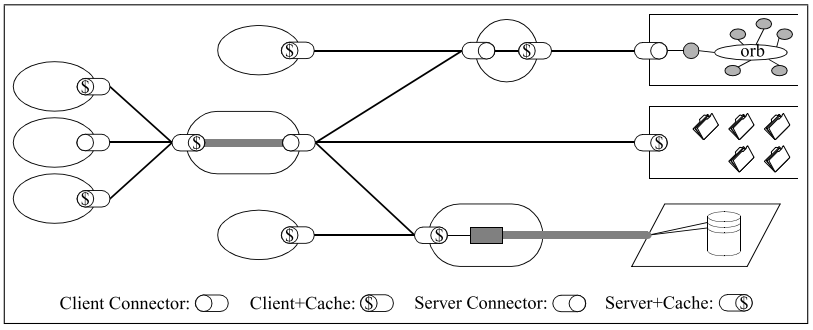

Figure 2.21: Uniform Layered Client Cache Stateless Server [28]

Code-On-Demand – The last constraint to be added is the code-on-demand style that allows client functionality to be extended by downloading and executing code in the form of applets or scripts. The presence of code-on-demand constraint is shown in Figure 2.22.

The REST Architectural View – Figure 2.23 shows an architectural (process) view of a REST-style

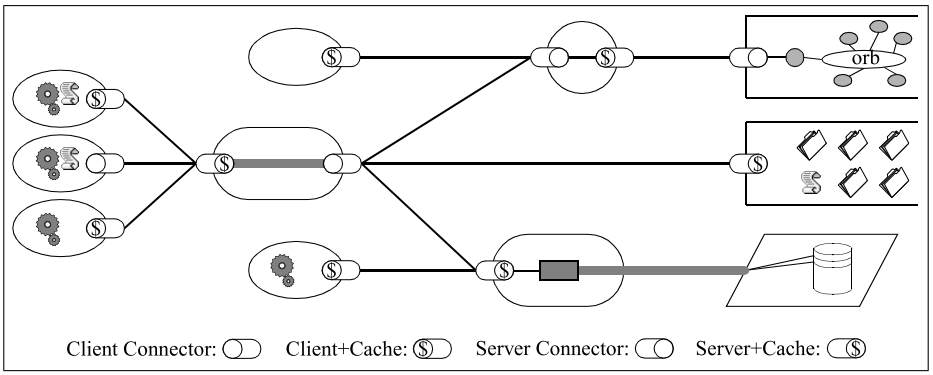

Figure 2.22: Code on Demand [28]

system highlighting different interaction relationships among components while processing three parallel requests (for example, a, b and c). This architectural view combines and contains all the REST constraints imposed at design time.

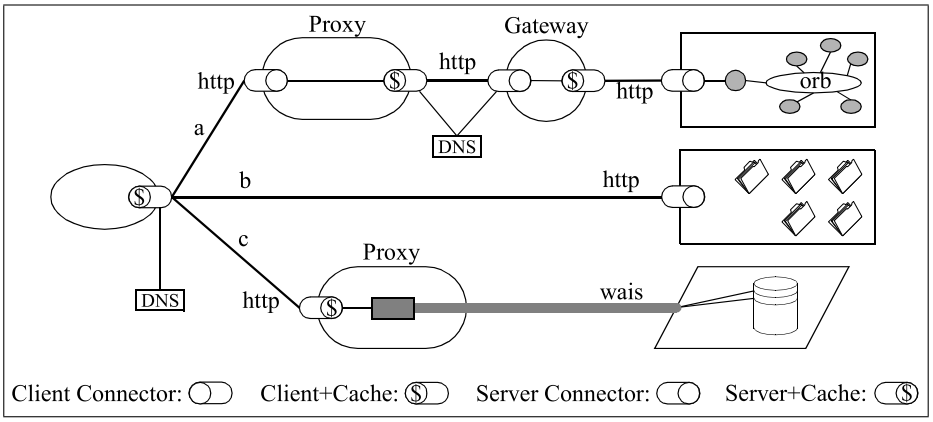

Figure 2.23: An REST Architectural Views [28]

Until recent, SOAP-based Web services built over RPC-style architecture were the mostly used approach for implementing an SOA. The RESTful approach to Web services is now a good choice due to its lightweight nature and the ability to transmit data directly over HTTP, hence the SOAP server is not required to handle SOAP messages any more. In the REST, every resource has an address. And, REST services usually do not involve WSDL definitions, even though there is WSDL 2.0 to describe REST services, which is not yet a *de facto* standard. A REST service is also platform and language -independent.

2.3 Detection of Service-oriented Antipatterns

As shown in Table 2.2, we can conclude that there is no research effort on detecting SOA antipatterns in SBSs, even though the adoption of SOA is significantly increasing. Table 2.2 shows the summary of the contributions that had been made on detecting different smells, design patterns and antipatterns in different software paradigms. Some of them have been proven very effective and widely accepted and adopted as well. However, clearly, it is worth to mention that there is no significant, not even preliminary contributions on detecting SOA antipatterns in SBSs, while the usage of SBSs and SOA architectural model is ever increasing. Figure 2.24 shows categories of antipatterns in different software paradigms.

What to Detect/Technology	OO	CBSs	SOA	Other
Code Smells	Khomh *et al.* [47] Maneerat and Muenchaisri [58] Moha *et al.* [64], Luo *et al.* [55] Emden and Moonen [26]	X	X	X
DS or DD or DF or Antipatterns	Khomh *et al.* [47, 48] Llano and Pooley [54] Moha *et al.* [64] Rao and Reddy [71] Salehie *et al.* [75] Kreimer [51] Marinescu [60, 61] Cortellessa *et al.* [24] Luo *et al.* [55], Biehl [10] Kessentini *et al.* [45] Settas *et al.* [76, 77] Choinzon and Ueda [21] Stoianov and Sora [78] Boussaidi *et al.* [12] Correa *et al.* [23]	Parsons and Murphy [68, 69] Chis [20]	X	Wong *et al.* [82] Fourati *et al.* [29]
Design Patterns	Stoianov and Sora [78] Heuzeroth *et al.* [41] Tsantalis *et al.* [80] Guéhéneuc and Antoniol [36] Kramer and Prechelt [50] Kaczor *et al.* [43, 44] Rasool and Mäder [72] Maggioni and Arcelli [56] Antoniol *et al.* [6]	X	X	X

Table 2.2: Summary of Contributions in the Literature for Detecting Smells, Design Patterns and Antipatterns (DS=Design Smells, DD=Design Defects, DF=Design Flaws, X=No contributions)

In Figure 2.24, each child of *Antipatterns* parent node represents different paradigms, and respective leafs represent corresponding technologies that are already studied. Whereas, hexagon with bold border represents service paradigm that is not studied yet, and hexagons with thinner border represents the corresponding SOA technologies that are still open to study. OO detection methods and tools (Section

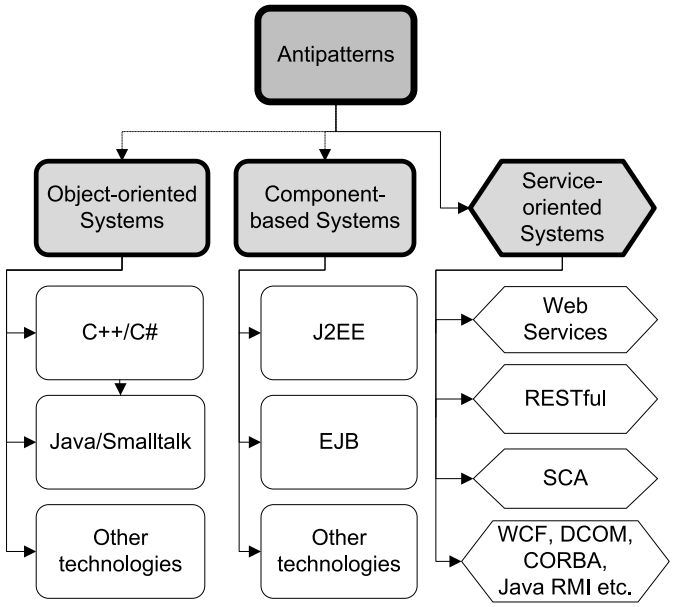

Figure 2.24: Categories of Antipatterns in Different Software Paradigms

2.1) cannot be directly applied to SOA. Indeed, SOA focuses on services as first-class entities whereas

OO focuses on classes, which are at the lower level of granularity. Moreover, the highly dynamic nature of an SOA environment raises several challenges (Section 3.2) that are not faced in OO development and requires dynamic analyses. However, all these previous works on OO systems and performance antipatterns detection form a sound basis of expertise and technical knowledge for building methods for the detection of SOA antipatterns.

Thus, we choose to fill this gap in the literature and put some contributions by means of a detection approach, an underlying detection framework that also supports the specification of SOA antipatterns, and even a tool for detecting SOA antipatterns. In the following section, we provide an applicable methodology and research plan for future in the direction of SOA antipattern detection.

Chapter 3

Thesis

With the background (Section 1.1) and motivation (Section 1.2) of this research and from the reviewed literature (Section 2.1) we identified some problems in the following on the detection of SOA antipatterns in SBSs. We also defined some objectives to solve those problems and proposed a solution (Section 3.5) in the line of those defined objectives. Following the above perspective the fundamental thesis of this work can be stated as follows:

Despite of the emergence of service-oriented architecture (SOA) and service-based systems (SBSs) in software development, no attention was given on detecting SOA antipatterns in service-based systems. Assuming that SOA antipatterns may hinder design, quality of service (QoS), and future evolution and maintenance of SBSs, their detection is important. By developing an approach supported by a framework for specifying and detecting SOA antipatterns, we intend to provide a solution to the problem of detecting SOA antipatterns in SBSs, along with the validation through experimental evidence.

3.1 Problems in the Literature

- **Problem 1: Impact of SOA Antipatterns is Not Yet Verified in SBSs** – The impact of OO antipatterns is already verified (i.e., Abbes *et al.* [4]) and has been shown the negative impact on program comprehension and on performance in general for OO systems. But there is no theoretical and empirical verification of the impact of SOA antipatterns within SBSs. We want to establish the relation between SOA antipatterns and their impacts, for example, (i) bad design and poor QoS criteria for an SBS, (ii) less understandability and analyzability (hence, the maintainability), and less changeability (hence, the evolvability).

- **Problem 2: No Specification of SOA antipatterns** – To detect SOA antipatterns, first one needs to specify them in a machine-processable and user-understandable format. Specification is also the initial step for the detection. Without the proper specification, SOA antipatterns may have ambiguous and dubious representation which are hard to use and describe. The present literature of SOA antipatterns is still in textual format (e.g., [18, 42, 49, 63]), sometimes without any description templates, which are hard to handle and use. Therefore, specifying SOA antipatterns is the primary step for our detection approach, and another open problem. The process of specification is integrated as a part of the detection approach and is performed within the underlying framework.

- **Problem 3: No Approach and Framework for the Detection of SOA Antipatterns** – SBSs operate in Internet-based dynamic environment. The execution context and dynamic nature of SBSs make the detection of SOA antipatterns challenging and an open problem. There are number of contributions [12, 23, 29, 45, 51, 55, 60, 61, 64, 68, 69, 75–78, 82] in the literature for the detection and correction of OO antipatterns and smells. However, there is no concrete method and technique for such detection in SBSs and to assess the QoS and design, in particular. Solving any software engineering problem using a framework-based approach increases the reusability and extensibility of the solution itself. Beyond the fact that there is no concrete approach for the detection of SOA antipatterns, also there is no generic framework for the detection of SOA antipatterns either. As the proof of concept, no tool is available either for detecting SOA antipatterns. Hence, we also intend to fill this gap by developing an autonomous tool.

3.2 Research Challenges

The key research challenges that we might face are:

(a) *Specifying SOA Antipatterns* – The current literature for SOA antipatterns is not matured enough and there are limited number of journals, proceedings and books on SOA antipatterns. Researchers and practitioners mostly depend on online resources, i.e, open forums, blogs, shared resources. Therefore, available references for specifying SOA antipatterns are not adequate.

(b) *Repository of SBSs* – The research on the detection of SOA antipatterns still did not gain much attention despite of its importance. And, there is no repository of SBSs to perform experiments, as the community did not make the effort to provide it. We can find some Web service search engines, e.g., *Seekda*[1] and *Woogle*[2], but no stand-alone and executable Service-based system with full design specification and implementation. Researchers are still in the lack of freely available SBSs as test bed.

(c) *Handling Dynamic Environment of SOA* – SBSs are developed using SOA design principles, and the execution environment of SBSs is dynamic involving many operating factors [27], e.g., composed services, interoperability among different systems and programming languages, services' abstraction level, statelessness of services, discoverability of services, granularity of services, location transparency of services (ability of a consumer to invoke a service) and many other issues are involved handling services in a SOA context. Thus, handling dynamic environment of SOA for analyzing SBSs is another major research challenge. *Service Unavailability* is another challenge to face. Client proxy may fail to connect to a Web service due to the fact that the Web service location may have changed or the service may not have been updated for a long time which is a very common problematic issue for any SBSs. Dynamically finding the new location of the same service requires some more development effort.

3.3 Scope of the Research

This research covers the area of service-oriented antipatterns [18, 25, 42, 49, 63]. Neither OO antipatterns, nor antipatterns related with other technologies or architectures, i.e., Dot Net, CORBA, J2EE, CBSs etc. are discussed in this proposal, as we are concerned only with service-based systems (SBSs) in the SOA context. Also, different smells, i.e., code smells [30] and design smells [30] of aforementioned technologies

[1] http://webservices.seekda.com/
[2] http://db.cs.washington.edu/woogle.html

or systems are out of the scope of this research. Hereby, SOA antipatterns [18, 25, 42, 49, 63] are the only scope of this research which still did not gain attention.

3.4 Objectives of the Thesis

3.4.1 General Objective of the Thesis

The main objective of this research is to propose an approach supported by a framework, towards the specification and detection of SOA antipatterns in SBSs, and to empirically validate the proposed approach.

3.4.2 Specific Objectives of the Thesis

The more specific objectives of this proposed research involves following phases:

1. To provide a complete and extensible Domain Specific Language (DSL) to specify SOA antipatterns with a higher abstraction of representation;
2. To provide a model driven engineering (MDE) methodology supporting from rule specification to context specific DSL generation;
3. To provide a formal verification of the impact of SOA antipatterns in an SBS.
4. To propose a generic approach (i.e., supporting different cutting edge SOA technologies) based on a framework for detecting SOA antipatterns in SBSs;
5. To develop a tool, e.g., in the form of an Eclipse plug-in, which is easy to install and able to perform the detection of SOA antipatterns in any SBSs;
6. To analyze and empirically validate the detection results.

Exploiting the above specific objectives will help this research to tackle the problems defined in the Section 3.1.

3.5 Proposed Solution

With the above context, the motivation and the problem definition, my PhD thesis will mainly focus on proposing a novel and innovative approach.

Our Approach: SODA – As the solution to the problem of specifying and detecting SOA antipatterns in SBSs, we propose an approach named as SODA (Service Oriented Detection for Antipatterns), for specifying and detecting SOA antipatterns. SODA is supported by a underlying framework, named as SOFA (Service Oriented Framework for Antipatterns), which helps to specify SOA antipatterns and to automatically generate detection algorithms from their specifications. We also state some research assumptions (Section 4.3.2) to verify, hence validating the SODA.

This approach is inspired from some previous OO smells and antipatterns detection approaches [20, 51, 60, 61, 64, 68, 82], which are not directly applicable to the context of SOA mainly due to two reasons: (i) their abstraction and granularity differences, for example, OO systems have *Classes* as their building blocks, whereas SBSs have *Services*, and (ii) analyses in OO systems are mainly static and some dynamic, but for SBSs its highly dynamic including some static analyses. Therefore, we

try to align our experiences from literature with OO antipatterns and their detection approaches to come up with a matured approach for detecting SOA antipatterns in SBSs. The approach consists of four main steps, from the analysis of the textual descriptions of SOA antipatterns through their reification to their detection in SBSs:

1. **Specification**: *This step consists of proposing a complete taxonomy including the terminology and a classification of SOA antipatterns to avoid inconsistency.* We intend to propose a specification process of SOA antipatterns to use them programmatically, i.e., parsing and building models, and store them in a repository of SOA antipatterns. This specification is based on an DSL that can represented using a Backus–Naur Form (BNF) grammar. The final outcome of this step are *rules cards* for antipatterns. Rule cards are high-level representation of SOA antipatterns using the DSL.

2. **Generation**: *This step aims at defining a procedure to generate detection algorithms automatically using a simple template-based technique.* From the previous step, using the specified rule cards we can generate detection algorithms for different SOA antipatterns. To generate detection algorithms, we can use *templates*, which are Java excerpts and have specific format for each SOA antipatterns. These *templates* have well-defined tags that are replaceable at execution time. Using the high-level representation of SOA antipatterns defined in the previous step, we can use our templates to visit and replace its tags with the values defined in the rule cards of SOA antipatterns, to generate detection algorithms. Finally, the generated algorithms are Java source code that are directly executable and applicable to a target SBS.

3. **Detection**: *At this step, we define a concrete technique supported by a framework for the detection of SOA antipatterns.* We intend to develop a framework to detect SOA antipatterns using the generated algorithms defined in the previous step. We name the proposed preliminary framework as SOFA (Service Oriented Framework for Antipatterns). This framework helps analysing and computing structural and behavioral properties of an SBS by means of *metrics*. Those metrics, including static and dynamic metrics enable us to formulate rules, and then the rule cards (see Section 4.1). The algorithms and proposed framework need to be generic to specify and detect any SOA antipatterns in any SBSs. By the term *generic*, we mean that an SBS can be designed and developed using different technologies, i.e., *Web services*, *SCA*, *SOAP/RPC*, *RESTful* or others, we intend to use all those SBSs with diverse technologies as our target SBSs for the detection.

4. **Validation**: *In this step, we aim to perform experimental studies on various SBSs to validate our detection results, and hence the approach and the framework* We intend to perform a number of experiments on different SBSs. After performing the detection in the previous step, which returns a number of suspicious service(s), a manual or statistical validation is required. We will statistically validate the detection results by using different statistical analyses, i.e., precision, recall and F-measure.

Ideally, these studies have to be performed separately by independent stakeholders. It is important that our method enables the subjects to specify different SOA antipatterns of different categories. We would like to compare also our approach in future, if some other approaches appear in the meantime. Before thoroughly investigating the problem, we also want to verify the impact of SOA antipatterns in SBSs (Problem 1 in Section 3.1).

Chapter 4

Methodology

Here, we detail our proposed methodology. Section 4.1 highlights the main steps of our approach, while Section 4.2 presents our underlying proposed framework. Finally, Section 4.3 provides the validation of our preliminary detection results.

4.1 The Approach: SODA

Figure 4.1 shows the proposed approach, SODA (Service Oriented Detection for Antipatterns), for detecting SOA antipatterns in SBS. Starting with the textual description of SOA antipatterns, from the specification and the generation of detection algorithms to the detection phase, we try to automate each step and follow through with a validation at the end. The 4 steps of the proposed approach include:

Step 1: Specification - It includes performing a thorough domain analysis by studying definitions, textual descriptions and specifications from the literature to identify relevant static and dynamic properties in the form of *metrics*, to specify SOA antipatterns. Then, using these intangible and measurable properties as the basis for the vocabulary to define our own DSL, and finally formalizing the rule cards. A rule card is the formal representation of a SOA antipattern at a high-level of abstraction and machine parsable. Figure 4.4 shows an example of rule cards for *Multi Service* and *Tiny Service*.

Step 2: Generation - From the rule cards of SOA antipatterns, we intend to generate detection algorithms automatically using a simple *template*-based technique.

Step 3: Detection - For the detection of SOA antipatterns, an underlying framework, SOFA is introduced. All the computations of static and dynamic metrics, i.e., identified relevant properties, and related analysis are performed in SOFA. It can also assist in semantic analysis of services. In this step, we apply the detection algorithms generated in the previous step on SBSs and report a number of suspicious services.

Step 4: Validation - The purpose of our validation is to demonstrate that the proposed approach is fit for its purpose, i.e., capable to detect SOA antipatterns with good accuracy and performance. The identified suspicious services in the previous step will be validated by means of different statistical measurements, i.e., precision, recall and F-measure. Also, the validation of the developed tool, can be performed by doing some user studies, i.e., controlled experiments, possibly followed by post-experimental questionnaire for both qualitative and quantitative measurements.

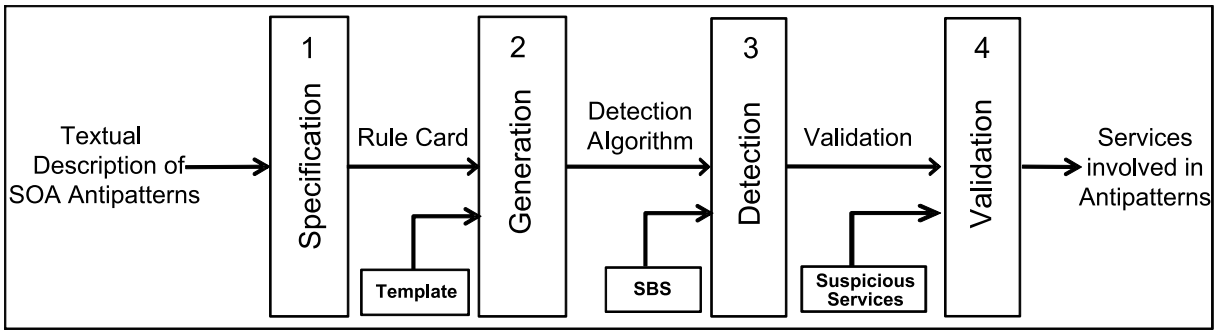

Figure 4.1: Proposed SODA Approach

4.1.1 Specification of SOA Antipatterns

Current literature does not formally specify any SOA antipatterns. For the sake of our proposed approach, we perform a domain analysis of SOA antipatterns by studying their definitions, textual descriptions and specifications in the current literature [25, 49, 73] and in online resources and articles [18, 42, 63]. This domain analysis allows us to identify properties relevant to SOA antipatterns, including static properties related to their design (e.g., cohesion and coupling) and also dynamic properties, such as QoS criteria (e.g., response time, availability and dynamic invocation). Static properties are properties that apply to the static descriptions of SBS, such as Web Service Description Language (WSDL) files, whereas dynamic properties are related to the dynamic behavior or nature of SBS as observed during their execution. We use these properties as the base vocabulary to define our own DSL, in the form of a rule-based language for specifying SOA antipatterns. The DSL offers software engineers high-level domain-related abstractions and variability points to express different properties of antipatterns depending on their own judgment and context. Figure 4.2 shows different high level steps for the specification of SOA antipattern.

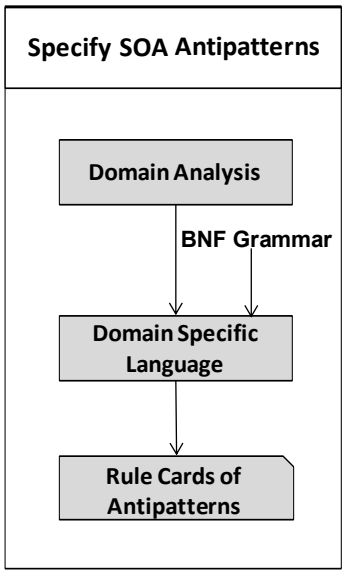

Figure 4.2: Specification of SOA Antipatterns

First, we perform a thorough domain analysis, and then using the BNF grammar, we define our DSL to specify the SOA antipatterns in the form of rule cards. Figure 4.4 shows an example of rule card for *Tiny Service* and *Multi Service*.

We specify SOA antipatterns using rule cards, i.e., sets of rules. We formalize rule cards with a BNF grammar, which determines the syntax of our DSL. Figure 4.3 shows the grammar used to express rule

```
1  rule_card     ::= RULE_CARD: rule_cardName { (rule)+ };
2  rule          ::= RULE: ruleName { content_rule };

3  content_rule ::= metric | relationship | operator ruleType (ruleType)+
4                 | RULE_CARD: rule_cardName

5  ruleType      ::= ruleName | rule_cardName

6  operator      ::= INTER | UNION | DIFF | INCL | NEG

7  metric        ::= id_metric ordi_value
8                  | id_metric comparator num_value
9  id_metric     ::= NMD | NIR | NOR | CPL | COH | ANP | ANPT | ANAM | ANIM
10                 | NMI | NTMI | RT | A
11 ordi_value    ::= VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW
12 comparator    ::= EQUAL | LESS | LESS_EQUAL | GREATER | GREATER_EQUAL

13 relationship  ::= relationType FROM ruleName cardinality TO ruleName cardinality
14 relationType  ::= ASSOC | COMPOS
15 cardinality   ::= ONE | MANY | ONE_OR_MANY | num_value NUMBER_OR_MANY

16  rule_cardName, ruleName, ruleClass ∈ string
17  num_value ∈ double
```

Figure 4.3: BNF Grammar of Rule Cards (Proposed Initial DSL) [65]

cards. A rule card is identified by the keyword `RULE_CARD`, followed by a name and a set of rules specifying this specific antipattern (Figure 4.3, line 1). A rule (lines 3 and 4) describes a metric, an association or composition relationship among rules (lines 13-15) or a combination with other rules, based on set operators including intersection, union, difference, inclusion, and negation (line 6). A rule can refer also to another rule card previously specified (line 4). A metric associates to an identifier a numerical or an ordinal value (lines 7 and 8). Ordinal values are defined with a five-point Likert scale: very high, high, medium, low, and very low (line 11). Numerical values are used to define thresholds with comparators (line 12), whereas ordinal values are used to define values relative to all the services of a SBS under analysis (line 11). We define ordinal values with the box-plot statistical technique [14] to relate ordinal values with concrete metric values while avoiding setting artificial thresholds. The metric suite (lines 9 and 10) encompasses both static and dynamic metrics. The static metric suite includes (but is not limited to) the following metrics: number of methods declared (`NMD`), number of incoming references (`NIR`), number of outgoing references (`NOR`), coupling (`CPL`), cohesion (`COH`), average number of parameters in methods (`ANP`), average number of primitive type parameters (`ANPT`), average number of accessor methods (`ANAM`), and average number of identical methods (`ANIM`). The dynamic metric suite contains: number of method invocations (`NMI`), number of transitive methods invoked (`NTMI`), response time (`RT`), and availability (`A`). Other metrics can be included by adding them to the SOFA framework [65]. Figure 4.4 illustrates the grammar with the rule cards of the *Multi Service* and *Tiny Service* antipatterns. The Multi Service antipattern is characterized by very high response time and number of methods, and low availability and

cohesion. A Tiny Service corresponds to a service that declares a very low number of methods and has a high coupling with other services. For the sake of clarity, we illustrate the DSL with two intra-service antipatterns, i.e., antipatterns within a service. However, the DSL allows also the specification of inter-service antipatterns, i.e., antipatterns spreading over more than one service. We provide the rule cards of such other more complex antipatterns later in the experiments section (see Section 4.3.1).

```
1 RULE_CARD: MultiService {
2  RULE: MultiService {INTER MultiMethod
3  HighResponse LowAvailability LowCohesion};
4  RULE: MultiMethod {NMD VERY_HIGH};
5  RULE: HighResponse {RT VERY_HIGH};
6  RULE: LowAvailability {A LOW};
7  RULE: LowCohesion {COH LOW};
8 };
```

(a) Multi Service

```
1 RULE_CARD: TinyService {
2  RULE: TinyService {INTER FewMethod
3                     HighCoupling};
4  RULE: FewMethod {NMD VERY_LOW};
5  RULE: HighCoupling {CPL HIGH};
6 };
```

(b) Tiny Service

Figure 4.4: Rule Cards for Multi Service and Tiny Service

Using a DSL offers greater flexibility than implementing ad hoc detection algorithms, because it allows describing antipatterns using high-level domain-related abstractions and focusing on *what* to detect instead of *how* to detect it [22]. Indeed, the DSL is independent of any implementation concern, such as the computation of static and dynamic metrics and the multitude of SOA technologies underlying SBS. Moreover, the DSL allows the adaptation of the antipattern specifications to the context and characteristics of the analyzed SBS by adjusting the metrics and associated values.

4.1.2 Generation of Detection Algorithms

From the specifications of SOA antipatterns described with our DSL, we automatically generate detection algorithms for each SOA antipattern. We implement the generation of the detection algorithms as a set of visitors on models of antipattern rule cards. The generation is based on templates and targets the services of the underlying framework described in the following subsection. Templates are excerpts of JAVA source code with well-defined tags. We use templates because the detection algorithms have common structures.

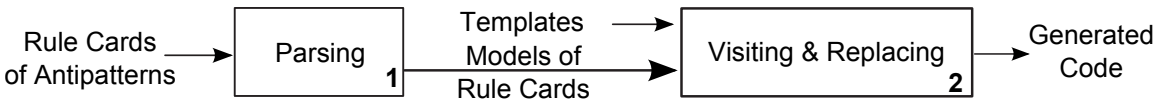

Figure 4.5: Generation of Detection Algorithms

Figure 4.5 sketches the different steps and artifacts of this generation process. First, rule cards of antipatterns are parsed and reified as models. Then, during the visit of the rule card models, the tags of templates are replaced with the data and values appropriate to the rules. The final source code generated for a rule card is the detection algorithm of the corresponding antipattern and this code is directly compilable and executable without any manual intervention.

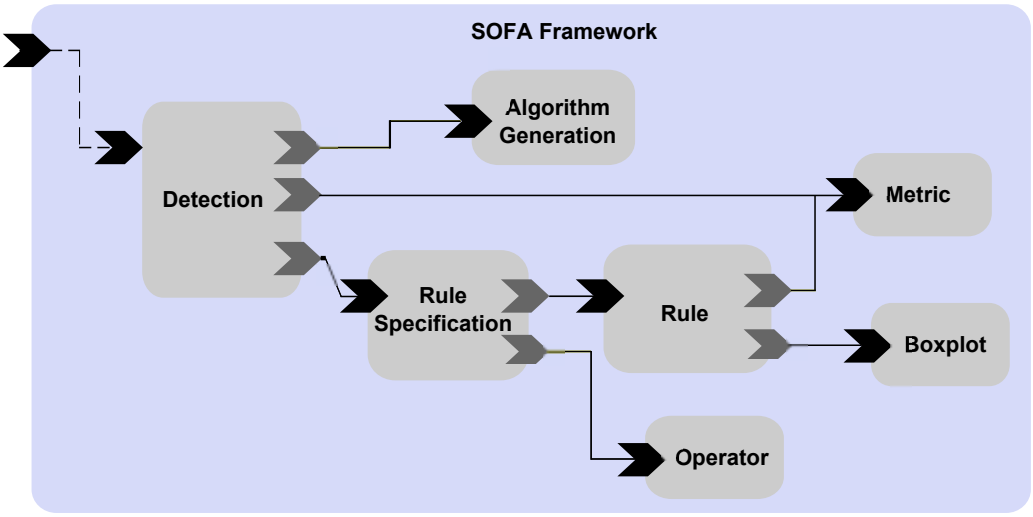

Figure 4.6: SOFA: Underlying Framework for the SODA Approach (black arrows represent service provided by the component, grey arrows dependency on referenced component) [65]

4.2 The Framework: SOFA

Figure 4.6 shows the underlying framework, SOFA (Service Oriented Framework for Antipatterns), that supports the specification and detection of SOA antipatterns in SBS. SOFA has seven modules, programmatically each of which represents a component providing a stand-alone service. The components include:

(1) *Detection Component*, representing the main detection engine that initiates and controls the overall detection process. It provides an interface to the client to run the detection process and help the client to visualize the detection results.
(2) *Metric Component* that provides the computation of both the static and dynamic metrics of the metric suite. This component also stores the static metric values in a repository to be used on the fly. Dynamic metrics cannot be restored, as they may change runtime.
(3) *Rule Specification Component* is responsible for specifying rule cards using the *Rule Component* and *Operator Component*. All the rule cards are also restored in a repository for being used by *Algorithm Generation Component*.
(4) *Algorithm Generation Component* generates the detection algorithms automatically from the specified rules. Then these detection algorithms will be executed by the clients using the *Detection Component*.
(5) *Rule Component* represents a repository of all the singleton rules that are composed of metrics, and depends on *Metric Component* to get required metrics.
(6) *Operator Component* provides all the boolean and comparison operators to merge or group the rules to form a rule card, and finally;
(7) *Boxplot Component* provides the means for computing boundary values and setting threshold values. It provides thus all kind of statistical analyses during detection phase of our approach.

The SOFA itself is a *service*-based framework and developed with the Service Component Architecture (SCA) technology [62].

4.3 Validation

4.3.1 Preliminary Experiments

To show the completeness and extensibility of our DSL, the accuracy of the generated algorithms, and the usefulness of the detection results with their related performance, we performed experiments with 10 SOA antipatterns on a service-based SCA [62] system, *Home-Automation*. This SBS has been developed independently for controlling remotely many basic household functions for elderly home care support. It includes 13 services with a set of 7 predefined scenarios (i.e., use cases) for executing them at runtime.

4.3.2 Assumptions

The experiments aim at validating the following initial set of assumptions:

A1. Generality: *The DSL allows the specification of many different SOA antipatterns, from simple to more complex ones.* This assumption supports the applicability of SODA using the rule cards on 10 SOA antipatterns, composed of 13 static and dynamic metrics.

A2. Accuracy: *The generated detection algorithms have a recall of 100%, i.e., all existing antipatterns are detected, and a precision greater than 75%, i.e., more than three-quarters of detected antipatterns are true positive.* Given the trade-off between precision and recall, we assume that 75% precision is significant enough with respect to 100% recall. This assumption supports the precision of the rule cards and the accuracy of the algorithm generation and of the SOFA framework.

A3. Extensibility: *The DSL and the proposed framework, SOFA is extensible for adding new SOA metrics and SOA antipatterns.* Through this assumption, we show how well the DSL, and in particular the new metrics, with the supporting SOFA framework, can be combined to specify and detect new antipatterns.

A4. Performance: *The computation time required for the detection of antipatterns using the generated algorithms is reasonably very low, i.e., in the order of few seconds.* This assumption supports the performance of the services provided by the SOFA framework for the detection of antipatterns.

A5. Technology: *The proposed approach, SODA supports any technologies, i.e., SBSs developed using different technologies, i.e., SOAP/RPC, SCA, REST, Web services etc.* This assumption supports that our proposed approach is capable of handling any SBSs implemented with diverse technologies, i.e., SOAP/RPC, SCA, REST, Web services, WCF etc. In particular, Web services is very popular, and REST is emerging and newly adopted technology.

4.3.3 Subjects

We apply our SODA approach using the SOFA framework to specify 10 different SOA antipatterns. Table 4.1 summarizes these antipatterns, of which the first seven are from the literature and three others have been newly defined, namely, the *Bottleneck Service*, *Service Chain*, and *Data Service*. These new antipatterns are inspired from OO code smells [30]. In these summaries, we highlight in bold the key concepts relevant for the specification of their rule cards given in Figure 4.7.

Multi Service also known as *God Object* corresponds to a service that implements a ***multitude of methods*** related to different business and technical abstractions. This aggregates too much into a single service, such a service is not easily reusable because of the ***low cohesion*** of its methods and is often ***unavailable*** to end-users because of its ***overload***, which may induce a ***high response time*** [25].

Tiny Service is a small service with ***few methods***, which only implements part of an abstraction. Such service often requires ***several coupled*** services to be used together, resulting in higher development complexity and ***reduced usability***. In the extreme case, a *Tiny Service* will be limited to ***one method***, resulting in many services that implement an overall set of requirements [25].

Sand Pile is also known as *'Fine-Grained Services'*. It appears when a service is ***composed*** by multiple smaller services sharing ***common data***. It thus has a ***high data cohesion***. The common data shared may be located in a ***Data Service*** antipattern (see below) [49].

Chatty Service corresponds to a set of services that exchange a lot of ***small data of primitive types***, usually with a ***Data Service*** antipattern. The *Chatty Service* is also characterized by a ***high number of method invocations***. *Chatty Service* chats a lot with each other [25].

The Knot is a set of ***very low cohesive*** services, which are ***tightly coupled***. These services are thus less reusable. Due to this complex architecture, the ***availability*** of these services can be ***low***, and their ***response time high*** [73].

Nobody Home corresponds to a service, defined but actually never used by clients. Thus, the ***methods*** from this service are ***never invoked***, even though it may be ***coupled*** to other services. But still they require deployment and management, despite of their no usage [42].

Duplicated Service a.k.a. *The Silo Approach* introduced by IBM corresponds to a set of ***highly similar*** services. Since services are implemented multiple times as a result of the silo approach, there may have ***common*** or ***identical methods*** with the ***same names and/or parameters*** [18].

Bottleneck Service is a service that is highly used by other services or clients. It has a ***high incoming and outgoing coupling***. Its ***response time*** can be ***high*** because it may be used by too many external clients, for which clients may need to wait to get access to the service. Moreover, its ***availability*** may also be low due to the traffic.

Service Chain a.k.a. *Message Chain* [30] in OO systems corresponds to a ***chain of services***. The *Service Chain* appears when clients request consecutive service invocations to fulfill their goals. This kind of ***dependency*** chain reflects the action of ***invocation*** in a ***transitive*** manner.

Data Service a.k.a. *Data Class* [30] in OO systems corresponds to a service that contains ***mainly accessor methods***, i.e., getters and setters. In the distributed applications, there can be some services that may only perform some simple information retrieval or ***data access*** to such services. *Data Services* contain usually ***accessor methods*** with ***small parameters*** of ***primitive types***. Such service has a ***high*** data ***cohesion***.

Table 4.1: List of Antipatterns (First Seven Antipatterns are Extracted from the Literature and Three Others are Newly Defined) [65]

4.3.4 Objects

We perform the experiments on two different versions of the *Home-Automation* system: the original version of the system, which includes 13 services, and a version modified by adding and modifying services

```
1 RULE_CARD: DataService {
2  RULE: DataService {INTER HighDataAccessor
3  SmallParameter PrimitiveParameter HighCohesion};
4  RULE: SmallParameter {ANP LOW};
5  RULE: PrimitiveParameter {ANPT HIGH};
6  RULE: HighDataAccessor {ANAM VERY_HIGH};
7  RULE: HighCohesion {COH HIGH};
8 };
```

(a) Data Service

```
1 RULE_CARD: TheKnot {
2  RULE: TheKnot {INTER HighCoupling
3  LowCohesion LowAvailability HighResponse};
4  RULE: HighCoupling {CPL VERY_HIGH};
5  RULE: LowCohesion {COH VERY_LOW};
6  RULE: LowAvailability {A LOW};
7  RULE: HighResponse {RT HIGH};
8 };
```

(b) The Knot

```
1 RULE_CARD: ChattyService {
2  RULE: ChattyService {
3  INTER TotalInvocation DSRuleCard};
4  RULE: DSRuleCard {RULE_CARD: DataService};
5  RULE: TotalInvocation {NMI VERY_HIGH};
6 };
```

(c) Chatty Service

```
1 RULE_CARD: NobodyHome {
2  RULE: NobodyHome {
3  INTER IncomingReference MethodInvocation};
4  RULE: IncomingReference {NIR GREATER 0};
5  RULE: MethodInvocation {NMI EQUAL 0};
6 };
```

(d) Nobody Home

```
1 RULE_CARD: BottleneckService {
2  RULE: BottleneckService {
3  INTER LowPerformance HighCoupling};
4  RULE: LowPerformance {
5  INTER LowAvailability HighResponse};
6  RULE: HighResponse {RT HIGH};
7  RULE: LowAvailability {A LOW};
8  RULE: HighCoupling {CPL VERY_HIGH};
9 };
```

(e) Bottleneck Service

```
1 RULE_CARD: SandPile {
2  RULE: SandPile {COMPOS FROM
3  ParentService ONE TO ChildService MANY};
4  RULE: ChildService {ASSOC FROM
5  ContainedService MANY TO DataSource ONE};
6  RULE: ParentService {COH HIGH};
7  RULE: DataSource {RULE_CARD: DataService};
8  RULE: ContainedService {NRO > 1};
9 };
```

(f) Sand Pile

```
1 RULE_CARD: ServiceChain {
2  RULE: ServiceChain {INTER TransitiveInvocation
3  LowAvailability};
4  RULE: TransitiveInvocation {NTMI VERY_HIGH};
5  RULE: LowAvailability {A LOW};
6 };
```

(g) Service Chain

```
1 RULE_CARD: DuplicatedService {
2   RULE: DuplicatedService {ANIM HIGH};
3 };
```

(h) Duplicated Service

Figure 4.7: Rule Cards for Different Antipatterns [65]

to inject intentionally some antipatterns. The modifications have been performed by an independent engineer to avoid biasing the results. Figure 4.9 shows the summary of the two versions of *Home-Automation*. Figure 4.8 shows the SCA design of the *Home-Automation* system. In Figure 4.8, each box represents a component providing a service, which also have references to other components. Details on the two versions of the system including all the scenarios and involved services are available online at http://sofa.uqam.ca.

4.3.5 Process

Using the SOFA framework, we generated the detection algorithms corresponding to the rule cards of the 10 antipatterns. Then, we applied these algorithms at runtime on the *Home-Automation* system using its set of 7 predefined scenarios.

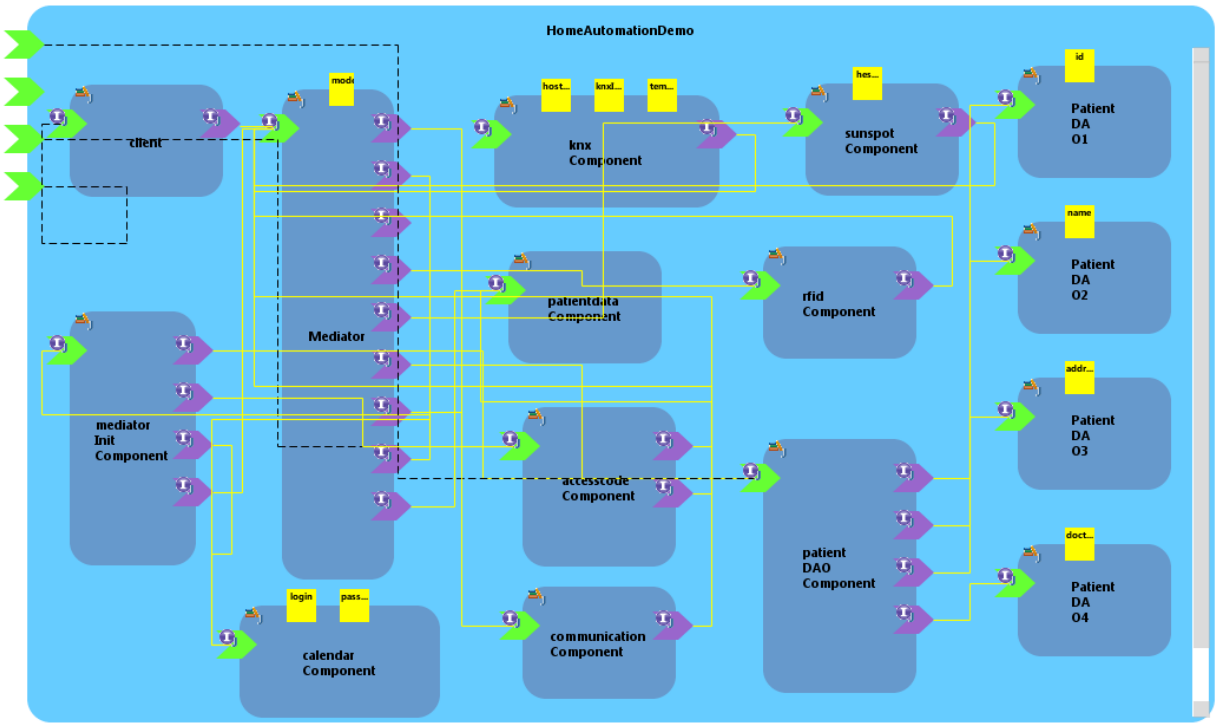

Figure 4.8: SCA Design of Home-Automation System

System Name	System Version	LOC	NOI	NOS	NOM	NOC
Home-Automation 1.0	original	3.2K	20	13	226	48
Home-Automation 1.1	evolved	3.4K	24	16	243	52

Figure 4.9: Objects: Two Versions of *Home-Automation* (LOC: Lines of Code, NOI: Number of Interface, NOS: Number of Service, NOM: Number of Method, NOC: Number of Class)

Finally, we validated the detection results by analyzing the suspicious services manually (1) to validate that these suspicious services are true positives, and (2) to identify false negatives (if any), i.e., missing antipatterns. For this last validation step, we use the measures of precision and recall [31]. Precision estimates the ratio of true antipatterns identified among the detected antipatterns, while recall estimates the ratio of detected antipatterns among the existing antipatterns.

$$precision = \frac{|\{existing\ antipatterns\} \cap \{detected\ antipatterns\}|}{|\{detected\ antipatterns\}|}$$

$$recall = \frac{|\{existing\ antipatterns\} \cap \{detected\ antipatterns\}|}{|\{existing\ antipatterns\}|}$$

This validation has been performed manually by two independent software engineers, whom we provided the descriptions of antipatterns and the two versions of the analyzed system *Home-Automation*.

$$F = 2 \times \frac{precision \times recall}{precision + recall}$$

We can also compute F-measure that is the harmonic mean of precision and recall, to conclude with a single value.

4.3.6 Preliminary Results

Table 4.2 presents the results for the detection of the 10 SOA antipatterns on the original and evolved version of *Home-Automation*. For each antipattern, the table reports the involved services in the second column, the version analyzed of *Home-Automation* in the third column, the analysis method: *static* (S) and–or *dynamic* (D) in the fourth, then the metrics values of rule cards in the fifth, and finally the computation times in the sixth. The two last columns report the precision and recall.

AntipatternName	ServicesInvolved	Version	Analysis	Metrics	DetectTime	Precision	Recall	F-measure
Tiny Service	[MediatorDelegate]	evolved	*S*	NOR: 4 CPL: 0.440 NMD: 1	0.194s	[1/1] 100%	[1/1] 100%	100%
Multi Service	[IMediator]	original	*S, D*	COH: 0.027 NMD: 13 RT: 132ms	0.462s	[1/1] 100%	[1/1] 100%	100%
Duplicated Service	[Communication-Service] [IMediator]	original	*S*	ANIM: 25%	0.215s	[2/2] 100%	[2/2] 100%	100%
Chatty Service	[PatientDAO] [IMediator]	original	*S, D*	ANP: 1.0 ANPT: 1.0 NMI: 3 ANAM: 100% COH: 0.167	0.383s	[2/2] 100%	[2/2] 100%	100%
Nobody Home	[UselessService]	evolved	*S, D*	NIR: >0 NMI: 0	1.154s	[1/1] 100%	[1/1] 100%	100%
Sand Pile	[HomeAutomation]	original	*S*	NCS: 13 ANP: 1.0 ANPT: 1.0 ANAM: 100% COH: 0.167	0.310s	[1/1] 100%	[1/1] 100%	100%
The Knot	[IMediator] [PatientDAO]	original	*S, D*	COH: 0.027 NIR: 7 NOR: 7 CPL: 1.0 RT: 57ms	0.412s	[1/2] 50%	[1/1] 100%	66.67%
Bottleneck Service	[IMediator] [PatientDAO]	original	*S, D*	NIR: 7 NOR: 7 CPL: 1.0 RT: 40ms	0.246s	[2/2] 100%	[2/2] 100%	100%
Data Service	[PatientDAO]	original	*S*	ANAM: 100% COH: 0.167 ANPT: 1.0 ANP: 1.0	0.268s	[1/1] 100%	[1/1] 100%	100%
Service Chain	[IMediator] [SunSpotService] [PatientDAO] [PatientDAO2]	original	*D*	NTMI: 4.0	0.229s	[3/4] 75%	[3/3] 100%	85.71%
AVERAGE					0.387s	[15/17] 92.5%	[15/15] 100%	95.24%

Table 4.2: Results for the Detection of 10 SOA Antipatterns in the Original and Evolved Version of *Home-Automation* System *(S: Static, D: Dynamic)* [65]

4.3.7 Details of the Results

We briefly present the detection results of the *Tiny Service* and *Multi Service*. The service `IMediator` has been identified as a *Multi Service* because of its very high number of methods (i.e., NMD is 13) and its low cohesion (i.e., COH is 0.027) compared to other Services in the target system. These metric values have been evaluated by the *Box-Plot* service respectively as high and low in comparison with the

metric values of other services of *Home-Automation*. For example, for the metric NMD, the *Box-Plot* estimates the median value of NMD in *Home-Automation* as equal to 2. In the same way, the detected *Tiny Service* has a very low number of methods (i.e., NMD is 1) and a high coupling (i.e., CPL is 0.44) with respect to other values. The values of the cohesion COH and coupling CPL metrics range from 0 to 1. In the original version of *Home-Automation*, we did not detect any *Tiny Service*. We then extracted one method from `IMediator` and moved it in a new service named `MediatorDelegate`, and then this service has been detected as a *Tiny Service*.

With the definition of 'Bottleneck Service' defined in rule card, it has a very high coupling with other services (both incoming references (NIR) and outgoing references (NOR)) and high response time. Where as, the median value for all other services were 0.5 for NOR and 0.0 for NIR, the *IMediator* service has a very high coupling, i.e., NIR and NOR is 7. We define the total coupling as the aggregation of NIR and NOR. And, it has also a very high response time, i.e. 40ms, due to too many usage. We therefore found *IMediator* also, as the 'Bottleneck Service'. The way, we define the rule card for 'The Knot', (a) it has a very high coupling value, i.e., 7 for NIR (whereas, the median for all the other services is 0.5), (b) very low cohesion among its methods, i.e., value 0.027 for COH, but we have the median of 0.167, and (c) a high response time, i.e., 57ms for the service with 'The Knot' antipattern. As expected, SOFA detects *IMediator* as 'The Knot'.

We also detected 4 other SOA antipatterns within the original version of *Home-Automation*, namely, *Duplicated Service*, *Chatty Service*, *Sand Pile*, and *Service Chain*. All these antipatterns involve more than one service, except *Duplicated Service*. Moreover, in the evolved version, we detected the *Nobody Home* antipattern, after an independent developer introduced the service `UselessService`, which is defined but never used in any scenarios. We detected a consecutive chain of invocations of `IMediator` → `SunSpotService` → `PatientDAO` → `PatientDAO2`, which forms a *Service Chain*, whereas engineers validated `IMediator` → `PatientDAO` → `PatientDAO2`. Therefore, we had the precision of 75% and recall of 100% for the *Service Chain* antipattern. Moreover, we detected the `HomeAutomation` itself as *Sand Pile*. Finally, an important point is that we use in some rule cards the dynamic property *Availability* (`A`). However, we did not report this value because it corresponds to 100% since the services of the system were deployed locally.

4.3.8 Discussion on the Assumptions

We have five assumptions to verify. Here for the preliminary experiment, we now verify first four assumptions stated in Section 4.3.2 on the basis of our preliminary detection results. We will verify remaining assumptions in our future experiments.

A1. Generality: *The DSL allows the specification of many different SOA antipatterns, from simple to more complex ones* [65]. Using our DSL, we specified 10 SOA antipatterns described in Table 4.1, as shown in rule cards given in Figure 4.4 and 4.7. These antipatterns range from simple ones, such as the *Tiny Service* and *Multi Service*, to more complex ones such as the *Bottleneck* and *Sand Pile*, which involve several services and complex relationships. In particular, *Sand Pile* has both the `ASSOC` and `COMPOS` relation type. Also, both *Sand Pile* and *Chatty Service* refer in their specifications to another antipattern, namely *DataService*. Thus, we show that we can specify from simple to more complex antipatterns, which supports the generality of our defined DSL.

A2. Accuracy: *The generated detection algorithms have a recall of 100%, i.e., all existing antipatterns are detected, and a precision greater than 75%, i.e., more than three-quarters of detected antipatterns are true positive* [65]. As indicated in Table 4.2, we obtain a recall of 100%, which means all existing antipatterns are detected, whereas the precision is 92.5%. We have high precision and recall because the

analyzed system, *Home-Automation* is a small SBS with 13 services. Also, the evolved version includes two new services. Therefore, considering the small *but* significant number of services and the well defined rule cards using DSL, we obtain such a high precision and recall. For the original *Home-Automation* version, out of 13 services, we detected 6 services that are responsible for 8 antipatterns. Besides, we detected 2 services (out of 15) that are responsible for 2 other antipatterns in the evolved system.

A3. Extensibility (DSL): *The DSL and the SOFA framework are extensible for adding new SOA antipatterns* [65]. The DSL has been initially designed for specifying the seven antipatterns described in the literature (see Table 4.1). Then, through inspection of the SBS and inspiration from OO code smells, we added three new antipatterns, namely the *Bottleneck Service*, *Service Chain* and *Data Service*. When specifying these new antipatterns, we reused four already-defined metrics and we added in the DSL and SOFA four more metrics (`ANAM`, `NTMI`, `ANP` and `ANPT`). The language is flexible in the integration of new metrics. However, the underlying SOFA framework should also be extended to provide the operational implementations of the new metrics. Such an addition can only be realized by skilled developers with our framework, that may require from 1 hour to 2 days according to the complexity of the metrics. Thus, by extending the DSL with these three new antipatterns and integrating them within the SOFA framework, we support A3.

A4. Performance: *The computation time required for the detection of antipatterns using the generated algorithms is reasonably very low, i.e., in the order of few seconds* [65]. We perform all experiments on an Intel Dual Core at 3.30GHz with 3GB of RAM. Computation times include computing metric values, introspection delay during static and dynamic analysis, and applying detection algorithms. The computation times for the detection of antipatterns is reasonably low, i.e., ranging from 0.194s to 1.154s with an average of 0.387s. Such low computation times suggest that SODA could be applied on SBS with larger number of services. Thus, we showed that we can support the fourth assumption positively.

4.3.9 Threats to Validity

The main threat to the validity of our results concerns would be their *external validity*, i.e., the possibility to generalize our approach to other SBSs. As future work, we plan to run experiments on other large SBSs. However, for this preliminary experiment, we considered two versions of *Home-Automation* to minimize such validity. For *internal validity*, the detection results not only depend on the services provided by the SOFA framework but also on the antipattern specifications using rule cards. We plan to performed experiments on a representative set of antipatterns to lessen this threat to the internal validity. The subjective nature of specifying and validating antipatterns is a threat to *construct validity*. We plan to lessen this threat by defining rule cards based on a literature review and thorough domain analysis and by involving independent engineers in the validation. We also plan to minimize *reliability validity* by automating the generation of the detection algorithms, such that each subsequent detection produce consistent sets of results with high precision and recall.

4.3.10 Remarks

In the preliminary experiment, we try to validate some of the assumptions defined in Section 4.3.2 in a small scale, i.e., with a small number of SOA antipatterns. Also, validating all the assumptions in a large scale remains as future work. A paper by Moha *et al.* was recently published and awarded the *second best* paper in ICSOC 2012 that contains the results presented in this proposal for validating first four assumptions in a small scale experiments. We will also verify other assumptions in our future experiments with large and complex SBSs. Before that, we want to verify the impact of SOA antipatterns in SBSs first.

Chapter 5

Research Plan

In this chapter, we present an overview of the state of the proposed research, what has already been done (Section 5.1, and the possible future directions for our research (Section 5.3) and Section 5.5). We also present a detailed plan for the possible publications to be produced from our research activities and obtained results.

5.1 Current State of the Research

Chapter 4 represents the current progress in the research towards the direction of SOA antipattern detection. We also have some short term (Section 5.3) and some long term (Section 5.5) plans for our research.

The phases that are already performed are:

- Studying the literature review and existing related research on the detection of OO smells, antipatterns and design patterns (Section 2.1), and SOA antipatterns (Section 2.3);
- Defining the first approach for specifying and detecting SOA antipatterns (Section 4.1);
  - Specifying a rule-based language and a initial framework for the detection of SOA antipatterns;
  - Automatic generation of detection algorithms for SOA antipatterns;
- Performing experimental studies for validating the approach on the detection (Section 4.3).

This research work contributes:

- To come up from a textual description of SOA antipatterns to a precise and structured description (i.e., in the form of rule cards);
- To build a repository of SOA antipatterns;
- To improve the quality, the evolution, and the maintenance of software by allowing detection of SOA antipatterns.

5.2 Current Contributions

Here is the list of our publications already published in different international conference and workshops.

Refereed Articles in International Conferences:

1. Naouel Moha, Francis Palma, Mathieu Nayrolles, Benjamin Joyen Conseil, Yann-Gaël Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel, *Specification and Detection of SOA Antipatterns*, ICSOC 2012, 10th International Conference on Service Oriented Computing, November 12-16, Shanghai, China, 2012. (2nd Best Paper Award)

Workshop and Symposium Papers:

1. Francis Palma, *Detection of SOA Antipatterns*, 8th PhD Symposium (Shanghai, China), in conjunction with ICSOC 2012, 10th International Conference on Service Oriented Computing, November 12-16, Shanghai, China, 2012.

2. Mathieu Nayrolles, Francis Palma, Naouel Moha and Yann-Gaël Guéhéneuc, *SODA: A Tool Support for the Detection of SOA Antipatterns*, ICSOC Demonstration Track (Shanghai, China), in conjunction with ICSOC 2012, 10th International Conference on Service Oriented Computing, November 12-16, Shanghai, China, 2012.

3. Francis Palma, Hadi Farzin, Yann-Gaël Guéhéneuc and Naouel Moha, *Recommendation System for Design Patterns in Software Development: An DPR Overview*, 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE'12), in conjunction with ICSE 2012, Zurich, Switzerland.

5.3 Plan for Short Term Work

Some of the steps described in the previous section need to be extended as follows:

- To specify more SOA antipatterns using predefined DSL in the form of rule cards;
- To extend the repository of detected and manually validated SOA antipatterns;
- To collaborate with other research groups of same interest to report the use of our method, to compute the precisions and recalls of the results, and to improve our specifications of the SOA antipatterns.

The phases for future research direction towards the completion of the thesis requirements are as follows:

- To provide a taxonomy (a complete and unified vocabulary in the form of DSL) for SOA antipatterns;
- To explore other possible ways of techniques towards the detection of SOA antipatterns;
- To ensure applicability of the proposed approach to other SOA technologies, to make it more generic to all SBSs. Currently all the experiments are performed on SCA-based services;
- To perform more experimental studies on the detection of SOA antipatterns.

5.4 Publication Plan

Figure 5.1 shows the graphical representation of our tentative research plan. Possible upcoming publications of the research results are planned to be published in the following conferences and journals. Each plan is described using an *Input-Process-Expected Output* template, where the *Input* refers to the input (i.e., objects) of the experiment, the *Process* is the methodology of the experiment that we intend to follow, and the *Expected Output* is the expected outcomes from the performed experiments.

- ***Publication 1*** – Being awarded the second best paper in ICSOC 2012, we were invited to extend our published work [65] to a journal version. Accordingly, we intend to perform one more thorough experiment followed by a rigorous analysis of the results. `(International Journal of Cooperative Information Systems (IJCIS), July-August 2013)`. We are currently going on with this plan.
  - *Input*: A couple of SBSs with different SOA technologies which may have SOA antipatterns in them.
  - *Process*: We plan to replicate the experiment in the paper by Moha *et al.* [65] with one more SBS, and validate the results. Such validation with more SBSs may help us to ensure the maturity of our approach.
  - *Expected Output*: Detected suspicious service(s) with statistical and manual validation.
- ***Publication 2*** – The Impact of SOA Antipatterns in Service-based Systems `(SOCA 2013, June)`. We plan to empirically show that SOA antipatterns have impact, i.e., bring bad consequences for SBSs resulting poor QoS measure.
  - *Input*: An SBS that may have SOA antipatterns.
  - *Process*: We plan to conduct empirical studies with a group of users, to see if an SBS with SOA antipatterns has: less – (i) comprehensibility, (ii) maintainability, and–or (iii) evolvability. Given an SBS, users may be asked to modify the existing design and adding new requirements after a comprehension task. Then, we can have qualitative and quantitative measurements based on the users' responses and experiences.
  - *Expected Output*: Different metric values of comprehensibility, maintainability and evolvability for the target SBS showing the impact of SOA antipatterns in SBS.
- ***Publication 3*** – A Metric Suite towards SOA Antipatterns Detection `(ICSOC 2014, December or ESOCC 2014, September)`. Currently, there is no metric suite, we intend to provide a complete metric suite towards the detection of SOA antipatterns.
  - *Input*: To provide a metric suite, we need: (i) an SBS for the experiment, (ii) a list of metrics with detail specifications to detect.
  - *Process*: We need to formally define a list of SOA metrics. Then, in an SBS, we want to calculate and report all the metric values. Through the *Publication 3*, we can provide a repository of SOA metrics.
  - *Expected Output*: A list of SOA metrics with their measured values.
- ***Publication 4*** – Detection of SOA Antipatterns in RESTful Services-based Systems `(ICIW 2014, March)`. We plan to replicate the proposed approach for RESTful Services to detect SOA antipatterns in the RESTful context. We also plan to investigate other SOA technologies, i.e., WSDL-based or SOAP/RPC-based Web services.

- *Input*: An REST-style SBS, rule cards and generated detection algorithms.
- *Process*: We plan to replicate the experiment in the paper by Moha *et al.* [65] with a REST-style SBS, and validate the results. Such validation with diverse SBSs will help us to have a generic approach and framework that support different SOA technologies.
- *Expected Output*: Detected suspicious RESTful service(s) with statistical and manual validation, and a list of possible SOA antipatterns.

- ***Publication 5*** – Building Rules using Formal Concept Analysis towards the Detection of SOA Antipatterns `(ICSOC 2013, December).` We already proposed an approach for detecting SOA antipatterns using manually predefined rule cards in our previous paper in ICSOC 2012. We plan to build the rules using a well-known technique, formal concept analysis (FCA) [9] to improve the detection precision. Using FCA, we can extract the concepts (as metrics) from the textual descriptions of the SOA antipatterns, and later use those concepts to formulate rule cards. We plan also to use other techniques for the specification and detection, i.e., a machine learning technique or different heuristics.
  - *Input*: For the input, we may have, (i) an SBS, (ii) rule cards built using the FCA technique, and, (iii) generated detection algorithms from previously built rule cards.
  - *Process*: We plan to replicate the experiment in the paper by Moha *et al.* [65] with an SBS, and validate the results. Unlike in the paper [65], where the rules and rule cards were built manually, we plan to build the rules using the well-known FCA technique. And, then generate the detection algorithms to detect SOA antipatterns.
  - *Expected Output*: Detected suspicious service(s) with statistical and manual validation, and a list of possible SOA antipatterns.

- ***Publication 6*** – Empirical Validation of the Proposed Approach and the Developed Detection Tool `(SOCA 2014, December).` Without empirical validation, all previous works from plan 1 to plan 5 cannot be formally established. We intend to perform a detail user study with our tool and analyze the results gathered.
  - *Input*: Academia or industrial users as *Subject*, the list of SOA antipatterns as the *Object*, and the proposed approach in the form of tool.
  - *Process*: We plan to conduct user study with our complete approach and the tool, to analyze and summarize detection results to arrive to a decisive conclusion.
  - *Expected Output*: An empirical evidence supporting the usefulness of the approach and the tool.

- ***Publication 7*** – Detection of SOA Antipatterns `(Journal of Systems and Software (JSS) 2015, June-July).` We plan to publish a journal with our complete approach and results with the validation summarizing all the findings from plan 1 to plan 6.

Thesis Writing – In parallel with the *Publication 7*, I plan to continue writing my Ph.D. thesis so that I can finish all my writings maximum by the Summer 2015.

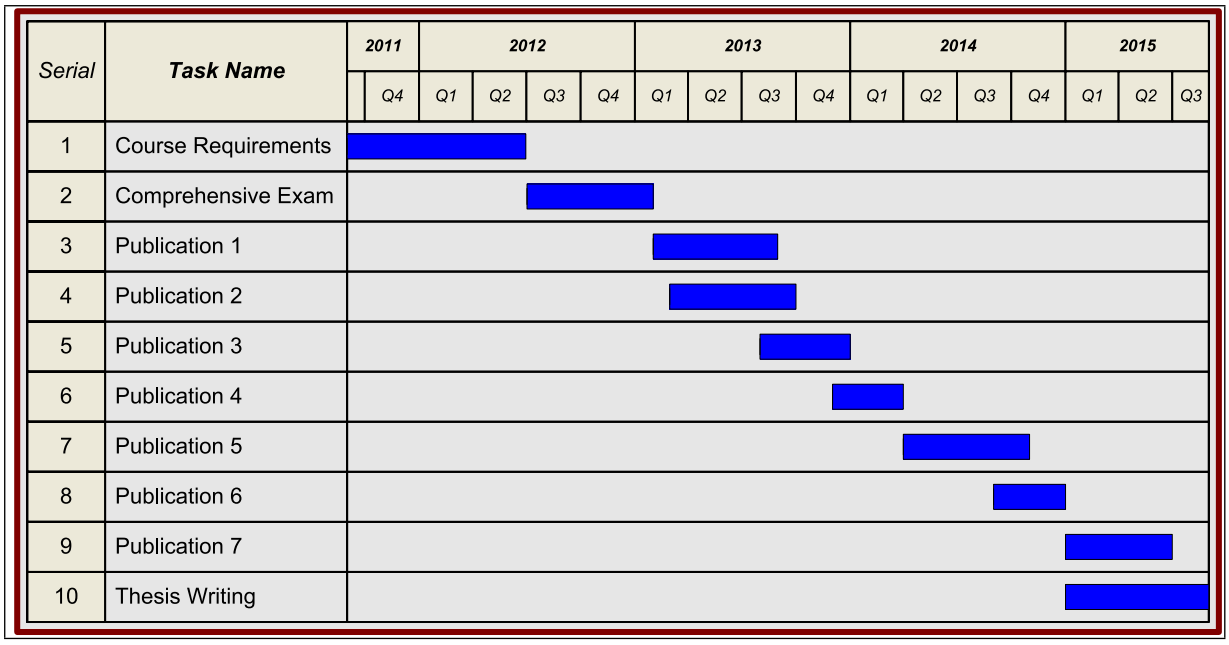

Figure 5.1: Short Time Research Plan: Timeline

5.5 Plan for Long Term Work

We have several long term research plans towards the detection of SOA antipatterns,

- We plan to conduct more experiments and analyze results using industrial setup with real systems. To perform such experiments, we need to perform:
  - Instrumenting the target real service-based system;
  - Deploy our framework within their (industrial partners') platform; and,
  - Finally, perform all the steps in our proposed approach to detect some antipatterns, and report suspicious service(s).

The experiment with industrial setup will help us to fine tune our approach and the tool more, and facilitate the approach and the tool to be more matured.

- We also plan to propose a corrective approach following the detection of SOA antipatterns as a long term research activity. To do that we intend to follow the steps:
  - To perform thorough literature review on the treatments of SOA antipatterns;
  - To propose an independent approach, possibly supported by a framework;
  - To perform the correction and validation by using the proposed approach in the previous step, of the corrected SOA antipatterns.

Through this research activity we will able to rectify the detected SOA antipatterns in SBSs.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The detection of service-oriented architecture (SOA) antipatterns is important to assess the design and QoS of SBSs and, to ease the maintenance and evolution of SBSs.

Several researchers have shown the impact of antipatterns in Object-oriented (OO) systems (Section 1.2.2). We also want to verify the impact of SOA antipatterns in SBSs. We provide a summary of the current contributions in the literature on smells, antipattern and design pattern detection (Section 2). We also reflect the summary in Table 2.2 to have a clear overview on the gaps in the current literature, i.e., no contributions over the detection of SOA antipattern in SBSs.

Indeed, there exists some problems in the current literature, for the detection of SOA antipatterns (Section 3.1) that we identified. To this end, we have defined some specific objectives for solving those existing problems (Section 3.4.2). With those objectives, we proposed a solution (Section 3.5) relying on a novel approach, Service Oriented Detection for Antipatterns – SODA (Section 4.1), supported by an underlying framework, Service Oriented Framework for Antipatterns – SOFA (Section 4.2). While trying to solve the problem of the detection of SOA antipatterns, we also face several research challenges (Section 3.2). One of our goal is to minimize or overcome those challenges.

For the validation of our proposed approach and first four research assumptions (Section 4.3.2) with a small scale experiment, we show some preliminary results (Sections 4.3.6) with an average detection precision of 92.5%, recall of 100%, and F-measure of 95.24%. Finally, we discuss our possible future research directions in Chapter 5 in detail.

In summary, our current contributions are: (1) specification of 10 common SOA antipatterns, (2) a first approach towards SOA antipattern detection, (3) an underlying framework supporting specification and detection of SOA antipattern, and, (4) some initial detection results for a small scale SBS. We also intend to: (1) perform a study showing the impact of SOA antipatterns in SBSs, (2) develop a complete tool supporting the specification and detection of SOA antipatterns in SBSs, (3) a concrete empirical evidence to show the effectiveness of the proposed approach and the tool, and finally, (4) verify all the research assumptions for the large scale SBSs.

6.2 Future Work

As the part of future work, we plan to validate other assumptions for the small scale SBSs (we already validated first four assumptions in our ICSOC 2012 paper [65]). We also plan to perform more experiments on large scale SBSs to validate all the assumptions again. Specifying more (in terms of number and complexity) SOA antipatterns in the form of rule cards is another future task.

For the short term plan, we want to carry out the following publication plans, described in Section 5.4:
- Publication 1: To extend the ICSOC 2012 paper to a journal version;
- Publication 2: To verify the impact of SOA Antipatterns in Service-based Systems;
- Publication 3: To propose a metric suite towards SOA antipattern detection;
- Publication 4: To perform detection of SOA antipatterns in RESTful, WSDL-based, and SOAP/RPC-based SBSs;
- Publication 5: To propose an approach for formulating rules using formal concept analysis towards the detection of SOA antipattern;
- Publication 6: To empirically validate the proposed approach and the developed detection tool;
- Publication 7: To summarize all the findings from previous steps and publish a journal article with complete results.

And for the long term plan, we want to conduct more experiments and analyze results using industrial setup with real service-based systems. We also plan to propose a corrective approach for the detected SOA antipatterns (Section 5.5) to advance the current research towards the correction of SOA antipatterns.

Bibliography

[1] Software Engineering – Product Quality – Part 1: Quality Model (First Edition, 2001-06-15). Reference Number: ISO/IEC 9126-1:2001(E)

[2] IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990 p. 1 (1990)

[3] Web Services: Principles and Technology (2008), http://books.google.ca/books?id=0qLX9nl87EwC

[4] Abbes, M., Khomh, F., Guéhéneuc, Y.G., Antoniol, G.: An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering. pp. 181–190. CSMR '11, IEEE Computer Society, Washington, DC, USA (2011)

[5] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications (2003), http://books.google.ca/books?id=CclkovBDqJkC

[6] Antoniol, G., Fiutem, R., Cristoforetti, L.: Using Metrics to Identify Design Patterns in Object-Oriented Software. In: Proceedings of the 5th International Symposium on Software Metrics. pp. 23–34. IEEE Computer Society Washington, DC, USA (1998)

[7] Arsanjani, A., Zhang, L.J., Ellis, M., Allam, A., Channabasavaiah, K.: Design an SOA Solution Using a Reference Architecture (March 2007), www.ibm.com/developerworks/library/ar-archtemp/

[8] Belady, L.A., Lehman, M.M.: A Model of Large Program Development. IBM Syst. J. 15(3), 225–252 (Sep 1976), http://dx.doi.org/10.1147/sj.153.0225

[9] BeLohlavek, R.: Introduction To Formal Concept Analysis

[10] Biehl, M.: APL - A Language for Automated Anti-Pattern Analysis of OO-Software (March 2006)

[11] Boehm, B.W.: Software Engineering Economics (1981)

[12] Boussaidi, G.E., Huynh, D.L., Moha, N.: Detection of Design Defects: Formal Concept Analysis and Metrics (December 2005)

[13] Brown, W.J., Malveau, R.C., III, H.W.M., Mowbray, T.J.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis (1998)

[14] Chambers, J.M., Cleveland, W.S., Tukey, P.A., Kleiner, B.: Graphical Methods for Data Analysis (1983)

[15] Channabasavaiah, K., Holley, K.: Migrating to a Service-oriented Architecture. IBM (April 2004), white paper

[16] Chappell, D.: Introducing SCA (July 2007), www.davidchappell.com/articles/introducing_sca.pdf

[17] Chatterjee, S., Webber, P.D.J.: Developing Enterprise Web Services: An Architect's Guide (2004), http://books.google.ca/books?id=LEpPzQ5mRDoC

[18] Cherbakov, L., Ibrahim, M., Ang, J.: SOA Antipatterns: The Obstacles to the Adoption and Successful Realization of Service-Oriented Architecture (November 2005), www.ibm.com/developerworks/webservices/library/ws-antipatterns/

[19] Chirila, C.: Instrument Software pentru Detectia Carentelor de Proiectare in Sisteme Orientate-Obiect. Ph.D. thesis (2001), diploma Thesis

[20] Chis, A.E.: Automatic detection of memory anti-patterns. In: Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications. pp. 925–926. OOPSLA Companion '08, ACM, New York, NY, USA (2008)

[21] Choinzon, M., Ueda, Y.: Detecting Defects in Object Oriented Designs Using Design Metrics. In: Proceedings of the 2006 conference on Knowledge-Based Software Engineering. pp. 61–72. IOS Press, Amsterdam, The Netherlands, The Netherlands (2006)

[22] Consel, C., Marlet, R.: Architecturing Software Using A Methodology for Language Development. Lecture Notes in Computer Science 1490, 170–194 (September 1998)

[23] Correa, A.L., Werner, C.M.L., Zaverucha, G.: Identification of Problematic Constructions in Object Oriented Applications: An Approach Based on Heuristics, Design Patterns and Antipatterns

[24] Cortellessa, V., Martens, A., Reussner, R., Trubiani, C.: A Process to Effectively Identify "Guilty" Performance Antipatterns. In: Rosenblum, D., Taentzer, G. (eds.) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 6013, pp. 368–382. Springer Berlin Heidelberg (2010)

[25] Dudney, B., Asbury, S., Krozak, J.K., Wittkopf, K.: J2EE AntiPatterns (2003)

[26] Emden, E.V., Moonen, L.: Java Quality Assurance by Detecting Code Smells. In: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02). WCRE, IEEE Computer Society, Washington, DC, USA (2002)

[27] Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design (2005)

[28] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis (2000), aAI9980887

[29] Fourati, R., Bouassida, N., Abdallah, H.: A Metric-based Approach for Anti-pattern Detection in UML Designs. In: Lee, R. (ed.) Computer and Information Science, Studies in Computational Intelligence, vol. 364, pp. 17–33. Springer Berlin Heidelberg (2011)

[30] Fowler, M.J., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code (1999)

[31] Frakes, W.B., Baeza-Yates, R.A.: Information Retrieval: Data Structures & Algorithms (1992)

[32] Frost, D.: Software Maintenance and Modifiability. In: Proc. 1985 Phoenix Conference on Computers and Communications, Phoenix, Az (1985)

[33] Gamma, E.: How to Use Design Patterns – A Conversation with Erich Gamma, part I (May 2005)

[34] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software (1994)

[35] Goth, G.: Software-as-a-Service: The Spark That Will Change Software Engineering? IEEE Distributed Systems Online 9(7) (2008)

[36] Guéhéneuc, Y.G., Antoniol, G.: DeMIMA: A Multilayered Approach for Design Pattern Identification. IEEE Transaction on Software Engineering 34(5), 667–684 (Sep 2008)

[37] Gurugé, A.: Web Services: Theory and Practice (2004), `http://books.google.ca/books?id=NzC06L8UWfsC`

[38] Hanna, M.: Maintenance Burden Begging for a Remedy. Datamation pp. 53–63 (1993)

[39] Harrison, W., Cook, C.: Insights on Improving the Maintenance Process through Software Measurement. In: International Conference on Software Maintenance (1990)

[40] Henderson, P., Yang, J.: Reusable Web Services. In: Proceedings of 8th International Conference, ICSR 2004. pp. 185–194 (2004)

[41] Heuzeroth, D., Holl, T., Högström, G., Löwe, W.: Automatic Design Pattern Detection. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension. IWPC, IEEE Computer Society, Washington, DC, USA (2003)

[42] Jones, S.: SOA Antipatterns (June 2006), `www.infoq.com/articles/SOA-anti-patterns`

[43] Kaczor, O., Guéhéneuc, Y.G., Hamel, S.: Efficient Identification of Design Patterns with Bit-vector Algorithm. In: Proceedings of the Conference on Software Maintenance and Reengineering. pp. 175–184. CSMR, IEEE Computer Society, Washington, DC, USA (2006)

[44] Kaczor, O., Guéhéneuc, Y.G., Hamel, S.: Identification of Design Motifs with Pattern Matching Algorithms. Information and Software Technology 52(2), 152–168 (2010)

[45] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design Defects Detection and Correction by Example. In: IEEE 19th International Conference on Program Comprehension (ICPC). pp. 81 –90 (june 2011)

[46] Khomh, F., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G.: An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. Empirical Software Engineering 17(3), 243–275 (2012)

[47] Khomh, F., Vaucher, S., Guéhéneuc, Y.G., Sahraoui, H.: A Bayesian Approach for the Detection of Code and Design Smells (August 2009)

[48] Khomh, F., Vaucher, S., Guéhéneuc, Y.G., Sahraoui, H.: BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns. Journal of Systems and Software 84(4), 559–572 (April 2011)

[49] Král, J., Žemlička, M.: Crucial Service-Oriented Antipatterns. vol. 2, pp. 160–171. International Academy, Research and Industry Association (IARIA) (2008)

[50] Kramer, C., Prechelt, L.: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Proceedings of the 3rd Working Conference on Reverse Engineering. WCRE, IEEE Computer Society, Washington, DC, USA (1996)

[51] Kreimer, J.: Adaptive Detection of Design Flaws. Electronic Notes on Theoratical Computer Science 141(4), 117–136 (Dec 2005)

[52] Kumar, B.V.: Web Services (2004), http://books.google.ca/books?id=cuif-nRRZfgC

[53] Lientz, B.P., Swanson, E.B.: Problems in Application Software Maintenance

[54] Llano, M.T., Pooley, R.: UML Specification and Correction of Object-Oriented Antipatterns. In: Proceedings of the 2009 Fourth International Conference on Software Engineering Advances. pp. 39–44. ICSEA, IEEE Computer Society, Washington, DC, USA (2009)

[55] Luo, Y., Hoss, A., Carver, D.L.: An Ontological Identification of Relationships between Antipatterns and Code Smells. In: Aerospace Conference, 2010 IEEE. pp. 1–10 (March 2010)

[56] Maggioni, S., Arcelli, F.: Metrics-based detection of micro patterns. In: Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics. pp. 39–46. WETSoM, ACM, New York, NY, USA (2010)

[57] Mahmood, Z.: Service Oriented Architecture: Potential Benefits and Challenges. In: Proceedings of the 11th WSEAS International Conference on Computers. pp. 497–501. ICCOMP'07, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA (2007), http://dl.acm.org/citation.cfm?id=1353956.1354045

[58] Maneerat, N., Muenchaisri, P.: Bad-smell Prediction from Software Design Model Using Machine Learning Techniques. In: 8th International Joint Conference on Computer Science and Software Engineering (JCSSE. pp. 331 –336 (May 2011)

[59] Mäntylä, M.V., Lassenius, C.: Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. Empirical Software Engineering 11(3), 395–431 (September 2006), http://dx.doi.org/10.1007/s10664-006-9002-8

[60] Marinescu, R.: Detecting Design Flaws via Metrics in Object-Oriented Systems. In: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39). TOOLS, IEEE Computer Society, Washington, DC, USA (2001)

[61] Marinescu, R.: Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance. pp. 350–359. ICSM '04, IEEE Computer Society, Washington, DC, USA (2004)

[62] Marino, J., Rowley, M.: Understanding SCA (Service Component Architecture) (2010), http://books.google.ca/books?id=KGHy7TX69UoC

[63] Modi, T.: SOA Management: SOA Antipatterns (August 2006), www.ebizq.net/topics/soa_management/features/7238.html

[64] Moha, N., Guéhéneuc, Y.G., Duchien, L., Meur, A.F.L.: DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transaction on Software Engineering 36(1), 20–36 (January 2010)

[65] Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.G., Baudry, B., Jézéquel, J.M.: Specification and Detection of SOA Antipatterns. 10th International Conference on Service Oriented Computing (November 2012)

[66] Nosek, J.T., Palvia, P.: Software Maintenance Management: Changes in the Last Decade. Journal of Software Maintenance 2(3), 157–174 (Sep 1990)

[67] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer 40(11), 38–45 (Nov 2007)

[68] Parsons, T., Murphy, J.: A Framework for Automatically Detecting and Assessing Performance Antipatterns in Component Based Systems Using Run-Time Analysis. In: The 9th International Workshop on Component Oriented Programming, part of ECOOP (2004)

[69] Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. Journal of Object Technology 7(3), 55–90 (April 2008)

[70] Pautasso, C., Zimmermann, O., Leymann, F.: Restful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In: Proceedings of the 17th international conference on World Wide Web. pp. 805–814. World Wide Web '08, ACM, New York, NY, USA (2008)

[71] Rao, A.A., Reddy, K.N.: Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix (2007)

[72] Rasool, G., Mader, P.: Flexible Design Pattern Detection based on Feature Types. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 243–252. ASE, IEEE Computer Society, Washington, DC, USA (2011)

[73] Rotem-Gal-Oz, A., Bruno, E., Dahan, U.: SOA Patterns (2012)

[74] Ryman, A.: Understanding Web Services (July 2003), `www.ibm.com/developerworks/websphere/library/techarticles/0307_ryman/ryman.html`

[75] Salehie, M., Li, S., Tahvildari, L.: A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In: Proceedings of the 14th IEEE International Conference on Program Comprehension. pp. 159–168. ICPC, IEEE Computer Society, Washington, DC, USA (2006)

[76] Settas, D., Meditskos, G., Bassiliades, N., Stamelos, I.G.: Detecting Antipatterns Using a Web-Based Collaborative Antipattern Ontology Knowledge Base. In: Salinesi, C., Pastor, O. (eds.) Advanced Information Systems Engineering Workshops, Lecture Notes in Business Information Processing, vol. 83, pp. 478–488. Springer Berlin Heidelberg (2011)

[77] Settas, D.L., Meditskos, G., Stamelos, I.G., Bassiliades, N.: SPARSE: A Symptom-based Antipattern Retrieval Knowledge-based System using Semantic Web Technologies. Expert Systems with Applications 38(6), 7633–7646 (2011)

[78] Stoianov, A., Sora, I.: Detecting Patterns and Antipatterns in Software using Prolog Rules. In: International Joint Conference on Computational Cybernetics and Technical Informatics (ICCC-CONTI). pp. 253–258 (may 2010)

[79] Tan, Y., Mookerjee, V.S.: Comparing Uniform and Flexible Policies for Software Maintenance and Replacement. IEEE Transaction on Software Engineering 31(3), 238–255 (Mar 2005)

[80] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design Pattern Detection Using Similarity Scoring. IEEE Transaction on Software Engineering 32(11), 896–909 (Nov 2006)

[81] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More (march 2005), `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0131488740`

[82] Wong, S., Aaron, M., Segall, J., Lynch, K., Mancoridis, S.: Reverse Engineering Utility Functions Using Genetic Programming to Detect Anomalous Behavior in Software. In: Proceedings of the 2010 17th Working Conference on Reverse Engineering. pp. 141–149. WCRE, IEEE Computer Society, Washington, DC, USA (2010)