Specification and Detection of SOA Antipatterns

Naouel Moha[1], Francis Palma[1,2], Mathieu Nayrolles[1,3],
Benjamin Joyen Conseil[1,3], Yann-Gaël Guéhéneuc[2],
Benoit Baudry[4], and Jean-Marc Jézéquel[4]

[1] Département d'informatique, Université du Québec à Montréal, Canada
moha.naouel@uqam.ca
[2] Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
{francis.palma, yann-gael.gueheneuc}@polymtl.ca
[3] CESI.eXia, Ecole Supérieur d'Informatique, France
{mathieu.nayrolles, benjamin.joyen-conseil}@viacesi.fr
[4] INRIA Rennes, Université Rennes 1, France
{bbaudry, jezequel}@irisa.fr

Abstract. Like any other complex software system, Service Based Systems (SBSs) must evolve to fit new user requirements and execution contexts. The changes resulting from the evolution of SBSs may degrade their design and quality of service (QoS) and may often cause the appearance of common poor solutions, called *Antipatterns*. Antipatterns resulting from these changes also hinder the future maintenance and evolution of SBSs. The automatic detection of antipatterns is thus important to assess the design and QoS of SBSs and ease their maintenance and evolution. However, methods and techniques for the detection of antipatterns in SBSs are still in their infancy despite their importance. In this paper, we introduce a novel and innovative approach supported by a framework for specifying and detecting antipatterns in SBSs. Using our approach, we specify 10 well-known and common antipatterns, including *Multi Service* and *Tiny Service*, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on *Home-Automation*, an SBS developed independently. This validation demonstrates that our approach enables the specification and detection of SOA antipatterns with the precision of more than 90% and the recall of 100%.

Keywords: Antipatterns, Service based systems, Specification, Detection, Quality of service, Design, Software evolution and maintenance.

1 Introduction

Service Oriented Architecture (SOA) [8] is an emerging architectural style that is becoming broadly adopted in industry because it allows the development of low-cost, flexible, and scalable distributed systems by composing ready-made services, *i.e.*, autonomous, reusable, and platform-independent software units that can be accessed through a network, such as the Internet. This architectural style can be implemented using a wide range of SOA technologies, such as OSGi, SCA, and Web Services. SOA allows building different types of Service Based

Systems (SBSs) from business systems to cloud-based systems. Google Maps, Amazon, eBay, PayPal, and FedEx are examples of large scale SBSs.

However, the emergence of such systems raises several challenges. Indeed, like any other complex software system, SBSs must evolve to fit new user requirements in terms of functionalities and Quality of Service (QoS). SBSs must also evolve to accommodate new execution contexts, such as addition of new devices, technologies, or protocols. All of these changes may degrade the design and QoS of SBSs, and may often result in the appearance of common poor solutions to recurring problems, called *Antipatterns*—by opposition to *design patterns*, which are good solutions to such problems that software engineers face when designing and developing systems. In addition to the degradation of the design and QoS, antipatterns resulting from these changes make it hard for software engineers to maintain and evolve systems.

Multi Service and *Tiny Service* are two common antipatterns in SBSs and it has been shown, in particular, that *Tiny Service* is the root cause of many SOA failures [15]. *Multi Service* is an SOA antipattern that corresponds to a service that implements a multitude of methods related to different business and technical abstractions. 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. 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 flexibility.

The automatic detection of such antipatterns is an important activity to assess the design and QoS of SBSs and ease the maintenance and evolution tasks of software engineers. However, few works have been devoted to SOA antipatterns, and methods and techniques for the detection of antipatterns in SBSs are still in their infancy.

Our goal is to assess the design and QoS of SBSs. To achieve this goal, we propose a novel and innovative approach (named SODA for Service Oriented Detection for Antipatterns) supported by a framework (named SOFA for Service Oriented Framework for Antipatterns) to specify SOA antipatterns and detect them automatically in SBSs. This framework supports the static and dynamic analysis of SBSs, along with their combination. Static analysis involves measurement of structural properties related to the design of SBSs, while dynamic analysis requires the runtime execution of SBSs for the measurement of runtime properties, mainly related to the QoS of SBSs. The SODA approach relies on the first language to specify SOA antipatterns in terms of metrics. This language is defined from a thorough domain analysis of SOA antipatterns in the literature. It allows the specifications of SOA antipatterns using high-level domain-related abstractions. It also allows the adaptation of the specifications of antipatterns to the context of the analyzed SBSs. Using this language and the SOFA framework dedicated to the static and dynamic analysis of SBSs, we generate detection algorithms automatically from the specifications of SOA antipatterns and apply them on any SBSs under analysis. The originality of our approach stems from

the ability for software engineers to specify SOA antipatterns at a high-level of abstraction using a consistent vocabulary and from the use of a domain-specific language for automatically generating the detection algorithms.

We apply SODA by specifying 10 well-known and common antipatterns and generating their detection algorithms. Then, we validate the detection results in terms of precision and recall on *Home-Automation*, an SBS developed independently by two Masters students. We consider two different versions of *Home-Automation*: (a) an original version, which includes 13 services, and (b) a version modified by adding and modifying services to inject intentionally some antipatterns. We show that SODA allows the specification and detection of a representative set of SOA antipatterns with a precision of 92.5% and a recall of 100%.

The remainder of this paper is organized as follows. Section 2 surveys related work on the detection of antipatterns in general, and in SBSs in particular. Section 3 presents our specification and detection approach, SODA, along with the specification language and the underlying detection framework, SOFA. Section 4 presents experiments performed on *Home-Automation* for validating our approach. Finally, Section 5 concludes and sketches future work.

2 Related Work

Architectural (or design) quality is essential for building well-designed, maintainable, and evolvable SBSs. Patterns and antipatterns have been recognized as one of the best ways to express architectural concerns and solutions. However, unlike Object Oriented (OO) antipatterns, methods and techniques for the detection and correction of SOA antipatterns are still in their infancy.

Unlike OO antipatterns, fewer books and papers deal with SOA antipatterns: most references are Web sites where SOA practitioners share their experiences in SOA design and development [4 ; 13 ; 17]. In 2003, Dudney *et al.* [7] 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. Most antipatterns described in this book cannot be detected automatically and are specific to a technology and correspond to variants of the Tiny and Multi Service. Another book by Rotem-Gal-Oz *et al.* [22] on SOA patterns and antipatterns will soon be published in Summer 2012. In a recent paper, Král *et al.* [15] described seven SOA antipatterns, which are caused by an improper use of SOA standards and improper practices borrowed from the OO design style. Other books exist on SOA patterns and principles [6 ; 9] that provide guidelines and principles characterizing "good" service oriented designs. Such books enable software engineers to manually assess the quality of their systems and provide a basis for improving design and implementation.

Several methods and tools exist for the detection [14 ; 16 ; 19 ; 24] and correction [1 ; 25 ; 26] of antipatterns in OO systems and various books have been published on that topic. For example, Brown *et al.* [2] introduced a collection of 40 antipatterns, Beck, in Fowler's highly-acclaimed book on refactoring [10],

compiled 22 code smells that are low-level antipatterns in source code, suggesting where engineers should apply refactorings. One of the root causes of OO antipatterns is the adoption of a procedural design style in OO system whereas for SOA antipatterns, it stems from the adoption of an OO style design in SOA system [15]. However, these OO detection methods and tools 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 that are not faced in OO development and requires dynamic analysis.

Other related works have focused on the detection of specific antipatterns related to system's performance and resource usage and–or given technologies. For example, Wong *et al.* [27] 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. In another relevant work, Parsons *et al.* [21] introduced an approach for the detection of performance antipatterns specifically in component-based enterprise systems (in particular, JEE applications) using a rule-based approach relying on static and dynamic analysis.

Although different, all these previous works on OO systems and performance antipattern detection form a sound basis of expertise and technical knowledge for building methods for the detection of SOA antipatterns.

3 The SODA Approach

We propose a three-step approach, named SODA, for the specification and detection of SOA antipatterns:
Step 1. Specify SOA antipatterns: This step consists of identifying properties in SBSs relevant to SOA antipatterns. Using these properties, we define a Domain-Specific Language (DSL) for specifying antipatterns at a high level of abstraction.
Step 2. Generate detection algorithms: In this step, detection algorithms are generated automatically from the specifications defined in the previous step.
Step 3. Detect automatically SOA antipatterns: The third step consists of applying, on the SBSs analyzed, the detection algorithms generated in *Step 2* to detect SOA antipatterns.

The following sections describe the first two steps. The third step is described in Section 4, where we detail experiments performed for validating SODA.

3.1 Specification of SOA Antipatterns

We perform a domain analysis of SOA antipatterns by studying their definition and specification in the literature [7 ; 15 ; 22] and in online resources and articles [4 ; 13 ; 17]. 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 and availability). Static properties are properties that apply to the static descriptions of SBSs, such as WSDL (Web Services Description Language) files, whereas dynamic properties are related to the dynamic behavior of SBSs as observed during their execution. We use these properties as the base vocabulary to define a 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.

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

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

 5  ruleType      ::= ruleName | ruleCardName

 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  ruleCardName, ruleName, ruleClass ∈ string
17  num_value ∈ double
```

Fig. 1. BNF Grammar of Rule Cards.

We specify antipatterns using rule cards, *i.e.*, sets of rules. We formalize rule cards with a Backus-Naur Form (BNF) grammar, which determines the syntax of our DSL. Figure 1 shows the grammar used to express rule 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 1, line 1). A rule (line 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 [3] to relate ordinal values with concrete metric values while avoiding setting artificial thresholds. The metric suite (lines 9-10) 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`),

```
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

Fig. 2. Rule Cards for Multi Service and Tiny Service

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.

Figure 2 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.*, services spreading over more than one service. We provide the rule cards of such other more complex antipatterns later in the experiments (see Section 4).

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 [5]. 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 SBSs. Moreover, the DSL allows the adaptation of the antipattern specifications to the context and characteristics of the analyzed SBSs by adjusting the metrics and associated values.

3.2 Generation of Detection Algorithms

From the specifications of SOA antipatterns described with the DSL, we automatically generate detection algorithms.

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 3 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

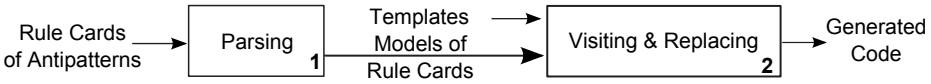

Fig. 3. Generation of Detection Algorithms.

rule card is the detection algorithm of the corresponding antipattern and this code is directly compilable and executable without any manual intervention.

This generative process is fully automated to avoid any manual tasks, which are usually repetitive and error-prone. This process also ensures the traceability between the specifications of antipatterns with the DSL and their concrete detection in SBSs using our underlying framework. Consequently, software engineers can focus on the specification of antipatterns, without considering any technical aspects of the underlying framework.

3.3 SOFA: Underlying Framework

We developed a framework, called SOFA (Service Oriented Framework for Antipatterns), that supports the detection of SOA antipatterns in SBSs. This framework, designed itself as an SBS and illustrated in Figure 4, provides different services corresponding to the main steps for the detection of SOA antipatterns (1) the automated generation of detection algorithms; (2) the computation of static and dynamic metrics; and (3) the specification of rules including different sub-services for the rule language, the box-plot statistical technique, and the set operators. The rule specification and algorithm generation services provide all

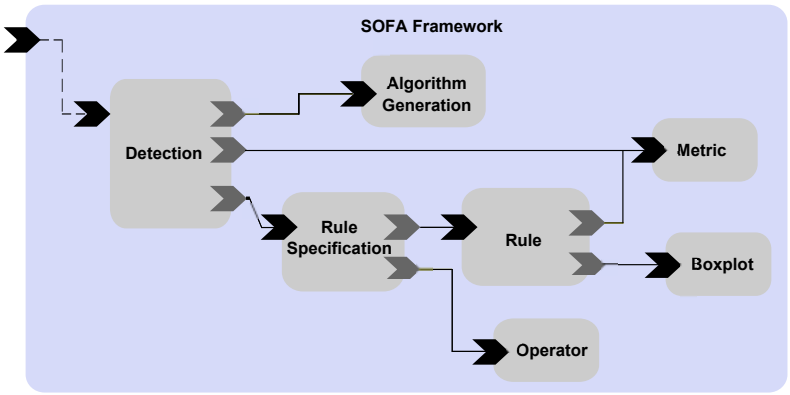

Fig. 4. The SOFA Framework.

constituents to describe models of rule cards as well as the algorithms to visit rule card models and to generate detection algorithms from these models. These different constituents rely on Model Driven Engineering techniques, which provide the means to define a DSL, parse it, and check its conformance with the grammar. We also use Kermeta [18], an executable metamodeling language, for generating the detection algorithms based on models of rule cards.

With respect to the computation of metrics, the generated detection algorithms call sensors and triggers implemented using the modules of the Galaxy framework [12]. These sensors and triggers, implemented as join points in an aspect-oriented programming style, allow, *at runtime*, the introspection of the interface of services and the triggering of events to add non-functional concerns, such as transactions, debugging, and, in our case, the computation of metrics such as response time.

We chose Kermeta and Galaxy for the sake of convenience because they are developed and maintained within our research team. Galaxy [12] is an open agile SOA framework supporting the SCA (Service Component Architecture) standard [20]. SCA is a relatively new standard, advocated by researchers and major software vendors, like IBM and Oracle, for developing technology agnostic and extensible SBSs. Galaxy encompasses different modules for building, deploying, running, and monitoring SBSs. Such a framework is essential for allowing the detection of SOA antipatterns at execution time through FraSCAti [23], which provides runtime support for the SCA standard. Furthermore, the SOFA framework is implemented itself as an SCA component to ease its use and evolution and to offer it as a service to end-users concerned by the design and QoS of their SBSs.

4 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 antipatterns on a service-based SCA 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 for executing it at runtime.

4.1 Assumptions

The experiments aim at validating the following four 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 14 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 SOFA framework are extensible for adding new SOA antipatterns.* Through this assumption, we show how well the DSL,

Table 1. List of Antipatterns. (The first seven antipatterns are extracted from the literature and three others are newly defined.)

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*** [7].
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 [7].
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) [15].
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 [7].
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*** [22].
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 [13].
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*** [4].
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* [10] 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* [10] 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***.

and in particular the 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.

4.2 Subjects

We apply our SODA approach using the SOFA framework to specify 10 different SOA antipatterns. Table 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 [10]. In these summaries, we highlight in bold the

key concepts relevant for the specification of their rule cards given in Figure 5.

```
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

Fig. 5. Rule Cards for Different Antipatterns

4.3 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 to inject intentionally some antipatterns. The modifications have been performed by an independent engineer to avoid biasing the results. Details on the two versions of the system including all the scenarios and involved services are available online at http://sofa.uqam.ca.

4.4 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. Finally, we validated the detection results by analyzing the suspicious services

manually to (1) validate that these suspicious services are true positives and (2) identify false negatives (if any), *i.e.*, missing antipatterns. For this last validation step, we use the measures of precision and recall [11]. Precision estimates the ratio of true antipatterns identified among the detected antipatterns, while recall estimates the ratio of detected antipatterns among the 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*.

4.5 Results

Table 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.

4.6 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 equal 13) and its low cohesion (*i.e.*, COH equal 0.027). 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 equal 1) and a high coupling (*i.e.*, CPL equal 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*.

We also detected 7 other antipatterns within the original version of *Home-Automation*, namely, *Duplicated Service*, *Chatty Service*, *Sand Pile*, *The Knot*, *Bottleneck Service*, *Data Service*, and *Service Chain*. All these antipatterns involve more than one service, except *Data Service* and *Duplicated Service*. The service `PatientDAO` has been detected as a *Data Service* because it performs simple data accesses. 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

Table 2. Results for the Detection of 10 SOA Antipatterns in the Original and Evolved Version of *Home-Automation* System *(S: Static, D: Dynamic)*

AntipatternName	ServicesInvolved	Version	Analysis	Metrics	DetectTime	Precision	Recall
Tiny Service	[MediatorDelegate]	evolved	*S*	NOR: 4 CPL: 0.440 NMD: 1	0.194s	[1/1] 100%	[1/1] 100%
Multi Service	[IMediator]	original	*S, D*	COH: 0.027 NMD: 13 RT: 132ms	0.462s	[1/1] 100%	[1/1] 100%
Duplicated Service	[Communication-Service] [IMediator]	original	*S*	ANIM: 25%	0.215s	[2/2] 100%	[2/2] 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%
Nobody Home	[UselessService]	evolved	*S, D*	NIR: >0 NMI: 0	1.154s	[1/1] 100%	[1/1] 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%
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%
Bottleneck Service	[IMediator] [PatientDAO]	original	*S, D*	NIR: 7 NOR: 7 CPL: 1.0 RT: 40ms	0.246s	[2/2] 100%	[2/2] 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%
Service Chain	[IMediator] [SunSpotService] [PatientDAO] [PatientDAO2]	original	*D*	NTMI: 4.0	0.229s	[3/4] 75%	[3/3] 100%
Average					0.387s	[15/17] 92.5%	[15/15] 100%

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.

4.7 Discussion on the Assumptions

We now verify each of the four assumptions stated previously using the detection results.

A1. Generality: *The DSL allows the specification of many different SOA antipatterns, from simple to more complex ones.* Using our DSL, we specified 10 SOA antipatterns described in Table 1, as shown in rule cards given in Figure 2 and 5. 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 complex antipatterns, which support the generality of our 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.* As indicated in Table 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: *The DSL and the SOFA framework are extensible for adding new SOA antipatterns.* The DSL has been initially designed for specifying the seven antipatterns described in the literature (see Table 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.* 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 SBSs with larger number of services. Thus, we showed that we can support the fourth assumption positively.

4.8 Threats to Validity

The main threat to the validity of our results concerns their *external validity*, *i.e.*, the possibility to generalize our approach to other SBSs. As future work, we plan to run these experiments on other SBSs. However, we considered two versions of *Home-Automation*. For *internal validity*, the detection results depend

on the services provided by the SOFA framework but also on the antipattern specifications using rule cards. We 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 try to lessen this threat by defining rule cards based on a literature review and domain analysis and by involving two independent engineers in the validation. We minimize *reliability validity* by automating the generation of the detection algorithm. Each subsequent detection produce consistent sets of results with high precision and recall.

5 Conclusion and Future Work

The specification and detection of SOA antipatterns are important to assess the design and QoS of SBSs and thus, ease the maintenance and evolution of SBSs. In this paper, we presented a novel approach, named SODA, for the specification and detection of SOA antipatterns, and SOFA, its underlying framework. We proposed a DSL for specifying SOA antipatterns and a process for automatically generating detection algorithms from the antipattern specifications. We applied and validated SODA with 10 different SOA antipatterns on an original and a evolved version of *Home-Automation*, a SBS developed independently. We demonstrated the usefulness of our approach and discussed its precision and recall.

As future work, we intend to enhance the detection approach with a correction approach to suggest refactorings and automatically, at runtime, correct detected SOA antipatterns, enabling software engineers to improve the design and QoS of their SBSs. Furthermore, we intend to perform other experiments on different SBSs from different SOA technologies, including SCA, Web Services, REST and EJB. The approach may require some adaptations from one technology to another because although SOA technologies share some common concepts and principles, they also have their own specific characteristics. Another targeted SBS is the SOFA framework itself since because this SBS will certainly evolve to handle various antipatterns and SBSs. We will thus ensure that the evolution of the SOFA framework does not introduce itself antipatterns.

Acknowledgments. The authors thank Yousri Kouki and Mahmoud Ben Hassine for their help with the implementation of *Home-Automation*. This work is partly supported by the NESSOS European Network of Excellence and a NSERC Discovery Grant. And, this work is in memory of Anne-Françoise Le Meur, our dearly departed colleague, who initiated the work.

References

1. Bart Du Bois, J.V., Demeyer, S.: Refactoring - Improving Coupling and Cohesion of Existing Code. In: Proceedings of the 11^{th} IEEE Working Conference on Reverse Engineering. pp. 144–151 (2004)
2. Brown, W., Malveau, R., McCormick III, H., Mowbray, T.: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons (1998)
3. Chambers, J., Cleveland, W., Tukey, P., Kleiner, B.: Graphical methods for data analysis. Wadsworth International (1983)

4. Cherbakov, L., Ibrahim, M., Ang, J.: SOA Antipatterns: The Obstacles to the Adoption and Successful Realization of Service-Oriented Architecture, www.ibm.com/developerworks/webservices/library/ws-antipatterns/
5. Consel, C., Marlet, R.: Architecturing Software Using A Methodology for Language Development. Lecture Notes in Computer Science 1490, 170–194 (September 1998)
6. Daigneau, R.: Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison-Wesley (November 2011)
7. Dudney, B., Asbury, S., Krozak, J., Wittkopf, K.: J2EE AntiPatterns. John Wiley & Sons Inc (2003)
8. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR (2005)
9. Erl, T.: SOA Design Patterns. Prentice Hall PTR (2009)
10. Fowler, M.J., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
11. Frakes, W.B., Baeza-Yates, R.A. (eds.): Information Retrieval: Data Structures & Algorithms. Prentice-Hall (1992)
12. Galaxy INRIA: The French National Institute for Research in Computer Science and Control, galaxy.gforge.inria.fr
13. Jones, S.: SOA Anti-patterns, www.infoq.com/articles/SOA-anti-patterns
14. Kessentini, M., Vaucher, S., Sahraoui, H.: Deviance From Perfection is a Better Criterion Than Closeness To Evil When Identifying Risky Code. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 113–122. ASE'10, ACM, New York, NY, USA (2010)
15. Král, J., Žemlička, M.: Crucial Service-Oriented Antipatterns. vol. 2, pp. 160–171. International Academy, Research and Industry Association (IARIA) (2008)
16. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer-Verlag (2006)
17. Modi, T.: SOA Management: SOA Antipatterns, www.ebizq.net/topics/soa_management/features/7238.html
18. Moha, N., Sen, S., Faucher, C., Barais, O., Jézéquel, J.M.: Evaluation of Kermeta for Solving Graph-based Problems. Journal on Software Tools for Technology Transfer 12(3-4), 273–285 (July 2010)
19. Munro, M.J.: Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In: Proceedings of the 11^{th} International Software Metrics Symposium. IEEE Computer Society Press (September 2005)
20. Open SOA: SCA Service Component Architecture - Assembly Model Specification (March 2007), www.osoa.org, version 1.00
21. Parsons, T., Murphy, J.: Detecting Performance Antipatterns in Component Based Enterprise Systems. Journal of Object Technology 7(3), 55–90 (April 2008)
22. Rotem-Gal-Oz, A., Bruno, E., Dahan, U.: SOA Patterns. Manning Publications Co. (2012), to be published in Summer 2012.
23. Seinturier, L., Merle, P., Fournier, D., Schiavoni, V., Demarey, C., Dolet, N., Petitprez, N.: FraSCAti - Open SCA Middleware Platform v1.4, frascati.ow2.org
24. 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 (June 2011)
25. Simon, F., Steinbruckner, F., Lewerentz, C.: Metrics Based Refactoring. In: Proceedings of the 5^{th} European Conference on Software Maintenance and Reengineering. pp. 14–16 (March 2001)
26. Trifu, A., Dragos, I.: Strategy-Based Elimination of Design Flaws in Object-Oriented Systems. In: Proceedings of the 4^{th} International Workshop on Object-Oriented Reengineering. Universiteit Antwerpen (July 2003)
27. 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 '10, IEEE Computer Society, Washington, DC, USA (2010)