Specification and Detection of SOA Antipatterns in Web Services

Francis Palma[1,2], Naouel Moha[2], Guy Tremblay[2], and Yann-Gaël Guéhéneuc[1]

[1] Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
{francis.palma, yann-gael.gueheneuc}@polymtl.ca
[2] Département d'informatique, Université du Québec à Montréal, Canada
{moha.naouel, tremblay.guy}@uqam.ca

Abstract. Service Based Systems, composed of Web Services (WSs), offer promising solutions to software development problems for companies. Like other software artefacts, WSs evolve due to the changed user requirements and execution contexts, which may introduce poor solutions—*Antipatterns*—may cause (1) degradation of design and quality of service (QoS) and (2) difficult maintenance and evolution. Thus, the automatic detection of antipatterns in WSs, which aims at evaluating their design and QoS requires attention. We propose SODA-W (Service Oriented Detection for Antipatterns in Web services), an approach supported by a framework for specifying and detecting antipatterns in WSs. Using SODA-W, we specify ten antipatterns, including *God Object Web Service* and *Fine Grained Web Service*, and perform their detection in two different corpora: (1) 13 weather-related and (2) 109 financial-related WSs. SODA-W can specify and detect antipatterns in WSs with an average precision of more than 75% and a recall of 100%.

Keywords: Antipatterns · Web Services · Specification · Detection

1 Introduction

Service Oriented Architecture (SOA) has already become the prevailing architectural style used in the industry [6]. SOA helps developing low-cost, reusable, and distributed business solutions by combining *services*, which are independent, portable, and interoperable program units that can be discovered and invoked through the Internet. In practice, SOA can be realised using various technologies and architectural styles including SCA (Service Component Architecture) [5], REST (REpresentational State Transfer), and Web services.

Web services is the leading SOA technology used nowadays to develop *Service-based systems* (SBSs) [15]. Amazon, Google, eBay, FedEx, PayPal, and many more companies, all leverage Web services. In the distributed systems literature, the term *Web service* is commonly used to refer to both SOAP-based and RESTful Web services. Nevertheless, in this paper, we focus on SOAP-based Web services because currently they are more widely adopted than those based on REST [15].

SBSs evolve to meet new requirements or to adapt to the changed execution contexts, *e.g.*, changes in transport protocols or in service contracts. Such changes may deteriorate the design and implementation, and worsen the QoS of Web services, and may cause the introduction of poor solutions, known as *Antipatterns*—in opposition to *design patterns* that are good solutions to recurring problems. In general, it has been shown that antipatterns negatively impact the evolution and maintenance of software systems [12].

God Object Web Service and *Fine Grained Web Service* are the two most common antipatterns in Web services [4]. The *God Object Web Service* describes a Web service that contains a large number of very low cohesive operations in its interface, related to different business abstractions. Being overloaded with a multitude of operations, a *God Object Web Service* may also have high response time and low availability. In contrast, *Fine Grained Web Service*, with few low cohesive operations, implements only a part of an abstraction. Such Web services often require several other coupled Web services to complete an abstraction, resulting in higher architectural complexity.

Despite the importance and extensive usage of Web services, no specification and automated approach for the detection of such antipatterns in Web services has been proposed. Such an approach to analyse the design and QoS of Web services and automatically identify antipatterns would help the maintenance and evolution of Web services. In fact, a few contributions have been made in the literature for the detection of SOA antipatterns in Web services including those in [14, 17, 18]. Yet, none of them provide the specification and all of them focus on the static analysis of Web service description files (*e.g.*, [17, 18]) or on antipatterns in other SOA technologies (*e.g.*, SCA [14]).

With the goal of assessing the design and QoS of Web services and filling the gap in the literature, we propose the SODA-W approach (Service Oriented Detection for Antipatterns in Web services) inspired from SODA [14]. SODA, supported by an underlying framework SOFA (Service Oriented Framework for Antipatterns), was the first approach dedicated to the specification and detection of antipatterns in SCA systems; it is, however, restricted to SCA. Instead, SODA-W is supported by an extended version of SOFA and is dedicated to the specification of SOA antipatterns and their automatic detection in Web services. The extended SOFA provides the means to analyse Web services statically, dynamically, or combining them. Static analyses refer to measuring the structural properties of Web services, whereas dynamic analyses invoke the real Web services and measure different properties, such as response time.

Therefore, the main contributions of this paper that leverage SODA-W are: (1) we add ten new metrics to our previous language proposed in [14] and adapt five other existing metrics in SOFA, (2) we specify ten Web service-specific antipatterns and perform the structural and semantic analysis of service interfaces, and finally (3) we perform detection for those ten antipatterns to validate SODA-W with more than 120 Web services in two different experiments. For the validation, we implement detection algorithms for the ten SOA antipatterns from their specifications, which we then apply on Web services. We perform the manual validation of the detection results in terms of precision, recall, and specificity.

Our results show that SODA-W allows to specify and detect SOA antipatterns with an average precision of more than 75% and a recall of 100%.

The remainder of this paper is organised as follows. Section 2 surveys related work on the detection of antipatterns, and in SBSs in particular. Section 3 lays out the approach, SODA-W, along with the language and the underlying framework, SOFA. Section 4 presents the experiments performed on Web services for validating SODA-W. Finally, Section 5 concludes and sketches future work.

2 Related Work

SOA antipatterns, Web service-specific antipatterns in particular, and their specification and detection are still in their infancy. A few books and articles address SOA antipatterns and most of the references are online [4, 13, 19]. Dudney *et al.* [4] first suggested a list of 52 antipatterns that are common in service-based architectures, and particularly in Web services. Antipatterns from that book are described informally. Rotem-Gal-Oz *et al.* [19] in their book listed some other SOA antipatterns also informally. In their paper, Král *et al.* [11] introduced seven SOA antipatterns that appear due to the improper use of SOA principles and standards. All the above works contributed to the existing catalogue of SOA antipatterns, but did not discuss their specification or detection.

A number of detection approaches [10,16,21] exist for object-oriented (OO) antipatterns. However, OO approaches are not applicable to the detection of SOA antipatterns because: (1) SOA is concerned with *services* as building blocks, whereas OO is concerned with *classes*, *i.e.*, services are coarser than classes in terms of granularity and (2) the highly dynamic nature of SOA compared to OO systems. Just a few works studied the detection of SOA antipatterns in Web services. Rodriguez *et al.* [18] performed detection for a set of Web service-specific antipatterns related to WSDL proposed by Heß *et al.* [9]. However, the primary focus of the work was not analysing or improving the design of Web services, rather on the WSDL writing conventions to improve their discoverability.

Moha *et al.* [14] proposed the SODA approach for specifying and detecting antipatterns in SCA systems (Service Component Architecture), relying on a rule-based language to specify antipatterns at a higher-level of abstraction than detection algorithms. In SODA, the detection algorithms are generated automatically and applied on SCA systems with a high accuracy. However, the proposed approach can only deal with local SCA components developed with plain Java and cannot handle remote Web services.

In another study, Rodriguez *et al.* [17] described *EasySOC* and provided a set of guidelines for service providers to avoid bad practices while writing WSDLs. Based on some heuristics, the authors detected eight bad practices in the writing of WSDL for Web services. The heuristics are simple rules based on pattern matching. The authors did not consider the design and QoS of the Web services and analysed the WSDL files statically. In this paper, instead, we analyse the Web services both statically and dynamically.

More recently, Coscia *et al.* [3] performed a statistical correlation analysis between a set of traditional code-level OO metrics and WSDL-level service metrics, and found a statistically significant correlation between them. Still, the main focus was not on identifying bad practices or poor design decisions in the service interfaces. Also, Sindhgatta *et al.* [22] performed a thorough literature survey on service cohesion, coupling, and reusability metrics, and proposed five new cohesion and coupling metrics, which they described as new quality criteria for service design. These metrics are even at the WSDL code-level; in contrast, we assess the design and QoS of Web services.

Given the above limitations in the literature, we try to come up with a viable solution for specifying and detecting SOA antipatterns in Web services.

3 Approach

We now describe the SODA-W (Service Oriented Detection for Antipatterns in Web services) approach dedicated to Web services (WSs). SODA-W involves three steps from the specification of Web service-specific antipatterns to their detection.
Step 1. Specification of SOA Antipatterns: We identify the relevant properties of Web service-specific antipatterns that we use to extend our previous domain-specific language (DSL) [14]. We then use this DSL to specify antipatterns.
Step 2. Generation of Detection Algorithms: This step involves the generation of detection algorithms from the specifications in the former step. In this paper, we performed this step manually by implementing concretely the algorithms in conformance with the rules specified in Step 1. We plan to automate this step.
Step 3. Detection of SOA Antipatterns: We apply the detection algorithms on a set of real WSs to detect antipatterns.

The following sections detail the first two steps. The last step is discussed in Section 4, where we perform the validation of SODA-W.

3.1 Specification of SOA Antipatterns

To specify SOA antipatterns, we performed a thorough domain analysis of antipatterns for WSs. We investigated their definitions and descriptions in the literature [4, 9, 11, 13, 18] because these mostly discussed WS-specific antipatterns. We identified a set of properties related to each antipattern, including static properties related to service design, *e.g.*, cohesion and coupling; and dynamic properties, *e.g.*, response time and availability. In general, static properties are recoverable from service interfaces. In contrast, dynamic properties are obtained by concretely invoking the WSs. We used these relevant properties to extend our DSL from [14]. Using this DSL, engineers can specify SOA antipatterns in the form of a rule-based language, using their own judgment and experience. A DSL allows engineers to focus on *what* to detect without being concerned about *how* to detect [2]. In fact, our DSL is implementation-independent, *i.e.*, it can be used regardless of the underlying technology of the system under analysis. However, the DSL needs to be extended for each new technology.

```
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    ::= ALS | ANIO | ANP | ANPT | ANAO | ARIP | ARIO | ARIM | CPL | COH | NCO
10                  |NOD | NOPT | NPT | NVMS | NVOS | RGTS
11                  |A | RT
12 ordi_value   ::= VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW
13 comparator   ::= < | ≤ | = | ≥ | >

14 rule_cardName, ruleName, ruleClass ∈ string
15 num_value ∈ double
```

Fig. 1: BNF grammar of rule cards for SODA-W.

The syntax of our DSL is shown in Figure 1 using a Backus-Naur Form (BNF) grammar. We apply a rule-based technique for specifying antipatterns, *i.e.*, each rule card combines a set of rules. The different constituents of our DSL are as follows: a *rule_card* is characterised by a name and a set of related rules (Figure 1, line 1). A *rule* (lines 3 and 4) is associated with a metric or it may combine other rules using different set operators (line 6) including intersection (INTER) or union (UNION). A rule can be a singleton rule or it can refer to another rule card (line 4). A metric may involve an ordinary value or it can have a comparator with a numeric value (lines 7 and 8). Ordinal values range from VERY_LOW to VERY_HIGH (line 12), and are used to define values compared to other candidate WSs under analysis. We use the box-plot statistical technique [1] to associate ordinal values with numeric values, to automatically set thresholds. Finally, the comparators include common mathematical operators (line 13).

Our metric suite (lines 9 to 11) includes both static (lines 9 and 10) and dynamic metrics (line 11). In [14], we had a set of 13 metrics defined for SCA domain. In this paper, we extend the DSL by adding ten new metrics specific to the domain of WSs as shown in Table 1. We also adapt some previously existing metrics (see Table 1). This adaptation is essential due to the non-trivial differences between SCA and WSs. For instance, SCA applications are built with *components*, while WSs use *services* as their first class entities. The other metrics remain the same as in [14] as noted in Table 1.

The ARIP, ARIO, and ARIM metrics combine both the structural and semantic similarity computation. Structural similarity uses the well-known Levenshtein Distance algorithm, whereas semantic similarity uses WordNet[3(a)] and CoreNLP[3(b)]. WordNet is a widely used lexical database that groups nouns, verbs, adjectives, etc. into the sets of synsets, *i.e.*, cognitive synonyms, each rep-

[3] (a) wordnet.princeton.edu (b) nlp.stanford.edu/software/corenlp.shtml

Table 1: The list of 19 metrics in SODA-W approach.

Metrics	Full Names	Versions
ALS	Average Length of Signatures	new
ARIP	Average Ratio of Identical Port-Types	new
ARIO	Average Ratio of Identical Operations	new
ARIM	Average Ratio of Identical Messages	new
NCO	Number of Crud Operations	new
NOPT	Number of Operations in Port-Types	new
NPT	Number of Port-Types	new
NVMS	Number of Verbs in Message Signatures	new
NVOS	Number of Verbs in Operation Signatures	new
RGTS	Ratio of General Terms in Signatures	new
ANP	Average Number of Parameters in Operations	adapted
ANPT	Average Number of Primitive Type Parameters	adapted
NOD	Number of Operations Declared	adapted
ANIO	Average Number of Identical Operations	adapted
ANAO	Average Number of Accessor Operations	adapted
CPL	Coupling	same
COH	Cohesion	same
A	Availability	same
RT	Response Time	same

```
1 RULE_CARD: GodObjectWebService {
2  RULE: GodObjectWebService {INTER
       LowCohesion MultiOperation
       HighRT LowA};
3  RULE: LowCohesion {COH VERY_LOW};
4  RULE: MultiOperation {NOD HIGH};
5  RULE: HighRT {RT VERY_HIGH};
6  RULE: LowA {A LOW};
7 };
```

(a) God Object Web Service

```
1 RULE_CARD: FineGrainedWebService {
2  RULE: FineGrainedWebService {INTER
             FewOperation HighCoupling
             HighCohesion};
3  RULE: FewOperation {NOD LOW};
4  RULE: HighCoupling {CPL VERY_HIGH};
5  RULE: HighCohesion {COH LOW};
6 };
```

(b) Fine Grained Web Service

Fig. 2: Rule cards for *God Object Web Service* and *Fine Grained Web Service*

resenting a distinct concept. We use WordNet to find the cognitive similarity between two (sets of) operations, messages, or port-types. We use Stanford's CoreNLP: (1) to find the base forms of a set of signatures of operations, messages, or port-types and (2) to annotate them with the part-of-speech (POS) tagger after we split the signatures based on the CamelCase.

Figure 2 shows the rule cards of the *God Object Web Service* [4] and *Fine Grained Web Service* [4] antipatterns as discussed in Section 1. A *God Object Web Service* (Figure 2(a)) is characterised by a high number of low cohesive operations and results in very high response time with low availability. A *Fine Grained Web Service* (Figure 2(b)) contains a fewer number of low cohesive operations with a high coupling resulting in higher development complexity. We also specify eight other WS-specific SOA antipatterns, whose rule cards are available in Section 4.

3.2 Generation of Detection Algorithms

The second step involves the implementation of the detection algorithms from the rule cards specified for each SOA antipattern. For each antipattern, we implement all the related metrics following its specification and write the detection algorithm in Java, which can directly be applied on any WSs. In the future, we will automate this algorithm generation process following a similar technique presented in [14].

3.3 Underlying Framework

We further develop the SOFA framework (Service Oriented Framework for Antipatterns) [14] to support the detection of SOA antipatterns in WSs. SOFA itself is developed as an SBS based on the SCA (Service Component Architecture) standards [5] and is composed of several SCA components. Figure 3 depicts the SOFA's key components: (1) *Rule Specification*—specifies rules relying on several other components, such as *Rule*, *Metric*, *Operator*, and *Boxplot*. The *Box-Plot* determines the ordinal values based on the numerical values computed for all the services under analysis; (2) *Algorithm Generation*—generates detection algorithms based on specified rules; and (3) *Detection*—applies detection algorithms generated in *Algorithm Generation* component on WSs.

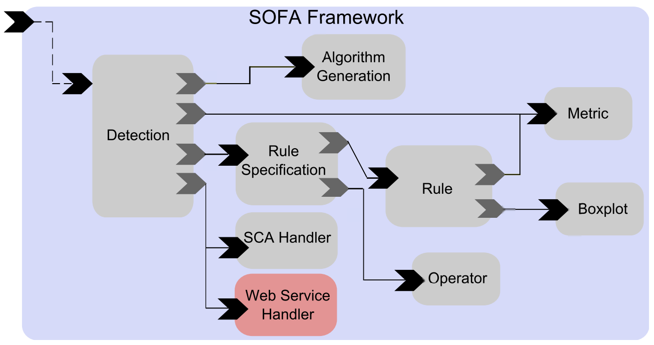

Fig. 3: The SOFA framework.

We added a new *Web Service Handler* component to the SOFA to allow the detection of Web service-specific antipatterns. The different functionalities performed by the *Web Service Handler* component include: (1) given *keywords*, it returns a list of WSs from a search engine, (2) it then filters broken service descriptions or unavailable services, and finally (3) for all WSs, it generates a list of SCA components. Concretely, these SCA components wrap WSs as our SOFA framework can only introspect SCA components.

We extended the SOFA framework by: (1) adding ten new Web service-specific metrics and (2) adapting five existing SCA-specific metrics. Combining those new and adapted metrics, we specify ten Web service-specific antipatterns as described in Figure 4 and perform their detection using SOFA. The addition of an antipattern requires the implementation of each metric following its specification. A metric can be reused for other antipatterns if they share the same metric in their specifications.

We use FraSCAti [20] as SOFA's runtime support. FraSCAti, itself developed as an SCA 1.1 application [5], provides a runtime environment for SCA applications. Being based on SCA, FraSCAti can provide component-based systems on top of diverse SOA technologies including Web services. In SOFA, we wrap each technology-specific services within an SCA component, thus providing a technology-agnostic platform to detect SOA antipatterns.

4 Validation

We want to show the completeness and the extensibility of our DSL, the preciseness of the detection algorithms, and the specificity of our rule cards. Therefore, we perform experiments with two sets of Web services (WSs) collected using a search engine: (1) 13 weather-related and (2) 109 finance-related WSs.

4.1 Hypotheses

We state three hypotheses that we want to examine in our experiments.
H1. Generality: *Our DSL allows the specification of various SOA antipatterns, from simple to more complex ones.* This hypothesis claims the applicability of our SODA-W approach that relies on metric-based (*i.e.*, 17 static and 2 dynamic metrics) rule cards for specifying ten Web service-specific SOA antipatterns.
H2. Accuracy: *The detection algorithms have an average precision of more than 75% and a recall of 100%, i.e., more than three-quarters of detected antipatterns are true positive and we do not miss any existing antipatterns.* Having a trade-off between precision and recall, we presume that 75% precision is acceptable while our objective is to detect all existing antipatterns, *i.e.*, 100% recall. We also show the specificity of the rule cards. This hypothesis claims the accuracy of the specified rule cards and the detection algorithms.
H3. Extensibility: *Our DSL and SOFA framework are extensible for adding new metrics and new SOA antipatterns.* In this hypothesis, we claim that the new metrics can be added and combined to specify new SOA antipatterns and that the SOFA framework can handle new antipatterns, including some specific to WSs, and detect them automatically.

4.2 Subjects

We specify ten different SOA antipatterns that are commonly found in WSs by applying our SODA-W approach. Figure 4 lists those Web service-specific SOA antipatterns. Among those ten antipatterns, eight are collected from the literature [4, 9, 11, 13, 18]. We also define two new antipatterns, namely *Duplicated Web Service* and *Data Web Service* inspired from OO antipatterns: *Silo Approach* and *Data Class*. Figure 4 emphasises the relevant properties of each antipattern in ***bold-italics***. Figure 5 shows the specifications of those antipatterns. We give concrete examples of those antipatterns and show how they manifest in practice on our site[4].

[4] http://sofa.uqam.ca/soda-w/

Ambiguous Name [18] is an antipattern where the developers use the names of ***interface elements*** (*e.g.,* ***port-types***, ***operations***, and ***messages***) that are ***very short*** or ***long***, include too ***general terms***, or even show the improper ***use of verbs***, etc. *Ambiguous names* are not ***semantically*** and ***syntactically*** sound and impact the ***discoverability*** and the ***reusability*** of a Web service.

Chatty Web Service [4] is an antipattern where a ***high*** number of ***operations*** are required to complete one abstraction where the ***operations*** are typically attribute-level ***setters*** or ***getters***. A chatty Web service may have many ***fine grained operations*** for which: (1) ***maintenance*** becomes harder since inferring the ***order of invocation*** is difficult and (2) ***many interactions*** are required, which ***degrades*** the overall ***performance*** with ***higher response time***.

CRUDy Interface [7] is an antipattern where the design encourages services the ***RPC-like behavior*** by creating ***CRUD-type operations***, *e.g.*, ***create_X()***, ***read_Y()***, etc. Interfaces designed in that way might be ***chatty*** because multiple operations need to be invoked to achieve one goal. In general, ***CRUD operations*** should ***not be exposed*** via ***interfaces***.

Data Web Service typically contains ***accessor operations***, *i.e.*, *getters* and *setters*. In a distributed environment, some Web services that may only perform some simple ***information retrieval*** or ***data access*** operations. A *Data Web Service* usually deals with ***very small messages*** of ***primitive types*** and may have ***high data cohesion***.

Duplicated Web Service, corresponds to a set of ***highly similar*** Web services. Because Web services are implemented multiple times as a result of the silo approach, there might exist ***common*** or ***identical operations*** with the ***same names and–or message parameters***.

Fine Grained Web Service [4] is a small Web service with ***few operations*** implementing only a part of an abstraction. Such a Web service often requires ***several coupled*** Web services to complete an abstraction, resulting in higher development complexity, ***reduced usability***. Moreover, since the ***related operations*** for an abstraction spread across services, individual services are ***less cohesive***.

God Object Web Service [4] corresponds to a Web service that contains a ***large number of operations*** related to different business abstractions. Often the client ***interactions break*** due to frequent changes in the Web service definition, hence cause ***low availability***. This antipattern affects the reusability because the operations are ***very low cohesive***. Moreover, being overloaded with a multitude of operations, this antipattern may also result in ***high response time***.

Low Cohesive Operations in the Same PortType [18] is an antipattern where developers place ***low cohesive operations*** in a ***single prototype***. From the Web services perspective, if the operations belonging to the same *prototype* do not provide a set of ***semantically related*** operations, the *prototype* becomes ***less cohesive***.

Maybe It's Not RPC [4] is an antipattern where the Web service mainly provides ***CRUD operations*** with a ***large number of parameters***. This antipattern causes ***poor system performance*** because the clients ***often wait*** for the synchronous ***responses***.

Redundant PortTypes [9] is an antipattern where ***multiple port-types*** are ***duplicated*** with the ***similar set*** of ***operations***. Very often, such *port-types* deal with the ***same messages***. The *Redundant PortType* antipattern may ***negatively impact*** the ***ranking*** of the Web Services.

Fig. 4: List of the ten SOA antipatterns in Web services.

4.3 Objects

Unlike open-source systems in OO, freely available real WSs are difficult to find for validating detection algorithms. There are some Web service search engines, like `eil.cs.txstate.edu/ServiceXplorer`, `programmableweb.com`, `myexperiment.org`, and `taverna.org.uk`, however, the number of such search engines is limited and often may not provide healthy service interface.

We perform experiments on two different sets of WSs collected from a Web service search engine, `programmableweb.com`. The first set includes 13 weather-related WSs (keyword *'Weather'*); and the second set includes 109 finance-related

```
RULE_CARD: AmbiguousName {
 RULE: AmbiguousName {INTER GeneralTerm
 ShortORLongSignature VerbedMessage
 MultiVerbedOperation};
 RULE: ShortORLongSignature {UNION
 ShortSignature LongSignature};
 RULE: LongSignature {ALS VERY_HIGH};
 RULE: ShortSignature {ALS VERY_LOW};
 RULE: GeneralTerm {RGTS HIGH};
 RULE: VerbedMessage {NVMS > 0};
 RULE: MultiVerbedOperation {NVOS > 1};
};
```

(a) Ambiguous Name

```
RULE_CARD: ChattyWebService {
 RULE: ChattyWebService {INTER LowCohesion
 HighDataAccessor MultiOperation
 LowPerformance};
 RULE: LowCohesion {COH LOW};
 RULE: HighDataAccessor {ANAO VERY_HIGH};
 RULE: MultiOperation {NOD HIGH};
 RULE: LowPerformance {INTER HighRT LowA};
 RULE: HighRT {RT HIGH};
 RULE: LowA {A LOW};
};
```

(b) Chatty Web Service

```
RULE_CARD: CRUDyInterface {
 RULE: CRUDyInterface {INTER ChattyInterface
 HighCRUDOperation};
 RULE: ChattyInterface {RULE_CARD:
 ChattyWebService};
 RULE: HighCRUDOperation {NCO > 1};
};
```

(c) CRUDy Interface

```
RULE_CARD: DataWebService {
 RULE: DataWebService {INTER HighCohesion
 PrimitiveParameter HighAccessor
 LowParameter};
 RULE: HighCohesion {COH HIGH};
 RULE: PrimitiveParameter {ANPT HIGH};
 RULE: HighAccessor {ANAO HIGH};
 RULE: LowParameter {ANP LOW};
};
```

(d) Data Web Service

```
RULE_CARD: DuplicatedWebService {
 RULE: DuplicatedWebService {INTER
 IdenticalPortType IdenticalOperation};
 RULE: IdenticalPortType {ARIP HIGH};
 RULE: IdenticalOperation {ARIO HIGH};
};
```

(e) Duplicated Web Service

```
RULE_CARD: LowCohesiveOperations {
 RULE: LowCohesiveOperations {INTER
 MultiOperation LowCohesivePT};
 RULE: MultiOperation {NOD HIGH};
 RULE: LowCohesivePT {ARIO LOW};
};
```

(f) Low Cohesive Operations

```
RULE_CARD: MaybeItsNotRPC {
 RULE: MaybeItsNotRPC {INTER HighRT
 HighCRUDOperation HighParameter};
 RULE: HighRT {RT HIGH};
 RULE: HighCRUDOperation {NCO VERY_HIGH};
 RULE: HighParameter {ANP HIGH};
};
```

(g) Maybe It's Not RPC

```
RULE_CARD: RedundantPortType {
 RULE: RedundantPortType {INTER
 MultiPortType MultiOps HighCohesivePT};
 RULE: MultiPortType {NPT > 1};
 RULE: MultiOps {NOPT > 1};
 RULE: HighCohesivePT {ARIP VERY_HIGH};
};
```

(h) Redundant *PortTypes*

Fig. 5: Rule cards for different SOA antipatterns in Web services.

WSs (keyword *'Finance'*). The complete list of all service interfaces that we experimented with is available online on our site[4].

4.4 Process

We specified the rule cards for ten Web service-specific antipatterns and implemented their detection algorithms using our SOFA framework. Then, we applied those algorithms on the WSs and reported any existing antipatterns. We manually validated the detection results to: (1) identify the true positives and (2) to find false negatives. The validation was performed by two students; we provided them with the descriptions of antipatterns and the service description file for each Web service along with its average response time. To measure the response time regardless of the network latency and physical location of a Web service, using the SAAJ[5(a)] standard implementation and SoapUI[5(b)], we arbitrarily invoked at least three operations from each real Web service, measured their response times, and took the average. We used precision and recall [8] to measure our detection accuracy. Precision concerns the ratio between the true detected antipatterns and all detected antipatterns, and recall is the ratio between the true detected antipatterns and all existing true antipatterns. Finally, we also calculate the specificity of our rule cards, *i.e.*, the ratio between all WSs identified as non-antipattern and total existing true negatives.

4.5 Results

Tables 2 and 3 present the detailed detection results for the ten SOA antipatterns. Each table reports the antipatterns in the first column followed by the involved WSs in the second. The third column shows the metric values for each Web service once it is identified as an antipattern. The fourth and fifth columns report the box-plot threshold values for each metric and the detection time for each antipattern, respectively. The last two columns show the precision (P) and recall (R) of our detection algorithms.

4.6 Details of the Results on 13 Weather Web services

We briefly explain the detection results obtained from the first experiment as presented in Table 2. We identified five WSs involved in four antipatterns, namely, *Ambiguous Name, Fine Grained Web Service, Low Cohesive Operations,* and *Redundant PortTypes*. For instance, the AIP3_PV_ImpactCallback in Table 2 is identified as an *Ambiguous Name* antipattern because this Web service offers operations with the signatures that (1) are very long (ALS=0.675), (2) use too many general terms (RGTS=0.85), (3) deal with many messages having verbs in their signatures (NVMS=26), and (4) have multiple verbs or action names (NVOS=7). In comparison to the median values, those values are high, *i.e.*, greater than the median but less or equal to the max. Therefore, we appropriately detected AIP3_PV_ImpactCallback as *Ambiguous Name* and had a precision and recall of 100% as confirmed by the manual validation.

We also detected SrtmWs-PortType, ShadowWs-PortType, and Hydro1KWs-PortType as *Fine Grained Web Service* antipatterns because they have very low values for NOD (*i.e.*, 2) and COH (*i.e.*, 0.0). As calculated by the *Box-Plot* component, the NOD values are low in comparison with the median of 5.5. Similarly, with only two operations defined, the cohesion values are not significant compared to other WSs, whose COH values are between 0.216 and 0.443. The manual validation revealed the correct identification of this antipattern for ShadowWs-PortType and Hydro1KWs-PortType. However, for the SrtmWs-PortType, the manual validation suggested that the operations defined in its service interface

[5] (a) saaj.java.net (b) www.soapui.org/

Table 2: Details on detection results for 13 Weather-related Web services.

Antipatterns	**Involved Web Services**	**Metrics**	**Boxplot Values** Min|Median|Max	**Detect Time**	**P**	**R**
Ambiguous Name	`AIP3_PV_Impact-Callback`	ALS 0.675 RGTS 0.85 NVMS 26 NVOS 7	0.027|0.463|0.675 0.0|0.0|0.85 4|6|54 1|3|20	0.69s	[1/1] 100%	[1/1] 100%
Chatty Web Service	*none detected*	*n/a*	*n/a*	300.23s	–	–
`CRUDy` Interface	*none detected*	*n/a*	*n/a*	244.48s	–	–
Data Web Service	*none detected*	*n/a*	*n/a*	1.03s	–	–
Duplicated Web Service	*none detected*	*n/a*	*n/a*	1.21s	–	–
Fine Grained Web Service	`SrtmWsPortType`	NOD 2 COH 0.0	2|5.5|27 0.0|0.216|0.443	1.04s	[2/3] 66.67%	[2/2] 100%
	`Hydro1KWsPortType`	NOD 2 COH 0.0	*same as above*			
	`ShadowWsPortType`	NOD 2 COH 0.0	*same as above*			
God Object Web Service	*none detected*	*n/a*	*n/a*	235.47s	–	–
Low Cohesive Operations	`ndfdXMLPortType`	NOD 12 ARIO 0.221	2|3|27 0.221|0.473|0.998	1.13s	[1/1] 100%	[1/1] 100%
May be It's Not `RPC`	*none detected*	*n/a*	*n/a*	235.47s	–	–
Redundant PortTypes	`AIP3_PV_Impact`	NOPT 3 ARIP 0.378	2|3|27 0.378|0.378|0.378	1.11s	[2/2] 100%	[2/2] 100%
	`AIP3_PV_Impact-Callback`	NOPT 9 ARIP 0.378	*same as above*			
Average				102.19s	[6/7] 85.71%	[6/6] 100%

could fulfill an abstraction, and did not consider `SrtmWs-PortType` as an antipattern. Thus, we have precision of 66.67% with 100% recall for this detection.

For this first experiment, our detection algorithms did not detect six other antipatterns (see Table 2).

4.7 Details of the Results on 109 Finance Web services

Table 3 shows the detail on each antipattern detected in the second experiment with 109 Finance-related `WSs`. We briefly describe here some antipatterns: `ForeignExchangeRates` and `TaarifCustoms` are both identified as the *Chatty Web Service* and *`CRUDy` Interface* antipatterns because of their low cohesion (`COH`$\approx$0.015), high average number of accessor operations (`ANAO` between 50 and 72.22), high number of operations (`NOD` between 9 and 24), and high response time (`RT` more than 3s), compared to other `WSs`. The box-plot values are shown in the corresponding rows for each metric. However, the manual analysis did not confirm `ForeignExchangeRates` as a *Chatty Web Service* because the order of invocation of the operations could easily be inferred from the service interface. The *`CRUDy` Interface* includes the rule card of *Chatty Web Service* in its specification. Therefore, the detection of `ForeignExchangeRates` as a *`CRUDy` Interface* was also not confirmed by the manual validation. Hence, we had the precision of 50% and recall of 100% for these two antipatterns.

We also identified `wsIndicadoresEconomicosHttpPost`, `wsIndicadoresEconomicosSoap`, and `wsIndicadoresEconomicosHttpGet` as *Redundant PortTypes*

Table 3: Details on detection results for 109 Finance-related Web services.

Antipatterns	**Involved Web Services**	**Metrics**	**Boxplot Values** Min|Median|Max	**Detect Time**	**P**	**R**
Ambiguous Name	`BLiquidity`	ALS 0.576 RGTS 0.682 NVMS 42 NVOS 7	0.013|0.226|0.81 0.0|0.613|0.75 1|64|482 0|6.5|48	1.02s	[8/8] 100%	[8/8] 100%
	`CurrencyServerWebService`	ALS 0.136 RGTS 0.682 NVMS 42 NVOS 5	*same as above*			
	... ...	... ...	... ...			
	`ProhibitedInvestors-Service`	ALS 0.158 RGTS 0.684 NVMS 12 NVOS 4	*same as above*			
Chatty Web Service	`ForeignExchangeRates`	COH 0.155 ANAO 50 NOD 24 RT 3286	0.0|0.25|0.667 0.0|0.961|100 1|12|70 172|1985|8592	1.89s	[1/2] 50%	[1/1] 100%
	`TaarifCustoms`	COH 0.116 ANAO 72.222 NOD 18 RT 4105	*same as above*			
`CRUDy` Interface	`ForeignExchangeRates`	COH 0.155 ANAO 66.667 NOD 9 RT 3113 NCO 9	0.0|0.25|0.667 0|0.96|100 1|11.5|70 172|1985|8592 0|9.5|62	1.81s	[1/2] 50%	[1/1] 100%
	`TaarifCustoms`	COH 0.103 ANAO 72.222 NOD 18 RT 4105 NCO 18	*same as above*			
Data Web Service	*none detected*	*n/a*	*n/a*	0.91s	–	–
Duplicated Web Service	*none detected*	*n/a*	*n/a*	1343.97s	–	–
Fine Grained Web Service	`XigniteTranscripts`	NOD 4 COH 0.125	1|12|70 0.0|0.25|0.667	0.85s	[2/2] 100%	[2/2] 100%
	`BGCantorUSTreasuries`	NOD 3 COH 0.083	*same as above*			
God Object Web Service	*none detected*	*n/a*	*n/a*	1.16s	–	–
Low Cohesive Operations	`ServiceSoap`	NOD 24 ARIO 0.253	1|12|70 0.0|0.435|1.0	242.49s	[7/7] 100%	[7/7] 100%
	`XigniteSecuritySoap`	NOD 25 ARIO 0.177	*same as above*			
	... ...	... ...	... ...			
	`XigniteSecurityHttpPost`	NOD 25 ARIO 0.177	*same as above*			
	`XigniteCorporate-ActionsSoap`	NOD 37 ARIO 0.268	*same as above*			
May be It's Not `RPC`	*none detected*	*n/a*	*n/a*	0.91s	–	–
Redundant PortTypes	`wsIndicadores-EconomicosHttpPost`	NOPT 2 ARIP 1.0	2|14|70 0.127|0.465|0.557	334.12s	[3/3] 100%	[3/3] 100%
	`wsIndicadores-EconomicosSoap`	NOPT 2 ARIP 1.0	*same as above*			
	`wsIndicadores-EconomicosHttpGet`	NOPT 2 ARIP 1.0	*same as above*			
Average				192.91s	[22/24] 91.67%	[22/22] 100%

antipattern with multiple identical port-types (*i.e.*, `NPT`>1 and `NOPT`>1) defined in their service interfaces, thus have `ARIP`=1.0, *i.e.*, a very high value compared to the median of 0.465. If a Web service has redundant *port-types*, it is a good practice to merge them, while making sure that this merge does not introduce a *God Object Web Service* antipattern. Seven other `WSs` were identified as *Low*

Cohesive Operations antipatterns (see Table 3), and two other WSs, *i.e.*, XigniteTranscripts and BGCantorUSTreasuries as *Fine Grained Web Service*. Both those WSs have a very small number of operations defined (NOD is 3 and 4) and have a low cohesion (COH between 0.083 and 0.125), compared to the maximum values (*i.e.*, 70 for NOD, and 0.667 for COH) from other WSs. Manual analysis also confirmed their detection, hence, we have precision and recall of 100% for *Redundant PortTypes* and *Fine Grained Web Service* antipatterns.

Again, for this experiment, we also did not identify four antipatterns on the set of 109 Finance-related WSs. As in Section 4.6 (see Table 2), we do not consider them to calculate the precision and recall. However, it is worth pointing out, the manual validation for 109 WSs is indeed a labor intensive task, and for each Web service it may take from 20 minutes to few hours based on the size of its interface.

4.8 Discussion on the Hypotheses

Following the results, we examine here three hypotheses stated in Section 4.1.

H1. Generality: In this paper, we specified ten WS-specific SOA antipatterns from the literature as shown in Figure 5 and described in Figure 4. We specified simpler antipatterns with fewer rules, such as *Low Cohesive Operations in the Same PortType* but also more complex antipatterns with composite rules, such as *CRUDy Interface* that is composed of another rule card, *i.e.*, *Chatty Web Service*. We also specified antipatterns combining six different rules, *Ambiguous Name* antipattern, for instance. Hence, this confirms our first hypothesis regarding the generality of our DSL. In fact, engineers can only use this DSL after analysing and integrating antipatterns properties to specify them.

H2. Accuracy: As shown in Tables 2 and 3, we obtained an average recall of 100% and an average precision of 88.69%. In the first experiment, with 13 WSs, we have a precision of 85.71%, whereas for the second experiment with 109 WSs, we have a precision and recall of 91.67% and 100%, respectively. Besides, we have the specificity of 98% for 13 WSs and 99% for 109 WSs. Thus, on average, we hold a precision of 88.69%, a recall of 100%, and a specificity 98.5%, which positively support our second hypothesis on the accuracy of our detection algorithms.

H3. Extensibility: We claim that our DSL and the SOFA framework are extensible for new antipatterns. In [14], we specified and detected ten antipatterns in SCA systems using our framework. In this paper, we specified and detected ten more Web service-specific antipatterns, and added them in the DSL and SOFA framework. More specifically, we added ten new metrics, such as NVMS, NOPT, RGTS, and NCO, etc. In addition, we added some variants of already existing metrics in the SOFA, *i.e.*, NOD, ANIO, ANAO, etc. Furthermore, we added new Web service-specific SOA antipatterns, such as *Low Cohesive Operations in the Same PortType*, *Maybe Its Not RPC*, and so forth. The designed language is flexible enough for integrating new metrics in the DSL. Our framework also supports the addition of new antipatterns through the implementation of new metrics and adaptation of existing ones to the new technology. This extensibility feature of our DSL and framework thus supports our third hypothesis.

4.9 Threats to Validity

As future work, we plan to generalise our findings to other large set of WSs. However, we tried to minimise the threat to the *external validity* of our results by performing two experiments with more than 120 WSs in two different domains. The detection results may vary based on the specification of the rule cards, and the way the components are implemented in the SOFA framework. *Internal validity* refers to the effectiveness of our approach and the framework. We made sure that the SOFA itself does not introduce antipatterns, to minimise the threat to the internal validity. Engineers may have different views and different levels of expertise on antipatterns, which may affect the specification of rule cards. We attempted to lessen the threat to *construct validity* by performing the specification of rule cards after a thorough literature review.

5 Conclusion

Web services are key artefacts for building Service-based systems. Like other systems, SBSs evolve due to new user requirements, which may lead to the introduction of antipatterns. The presence of SOA antipatterns may hinder software maintenance and evolution. This paper presented the SODA-W approach (Service Oriented Detection for Antipatterns in Web services) to specify and detect SOA antipatterns in Web services. Detection of antipatterns in Web services requires an in-depth analysis of their design, implementation, and QoS.

We applied SODA-W to specify ten common SOA antipatterns in Web services domain. Using an extended SOFA framework (Service Oriented Framework for Antipatterns), in an extensive validation with ten SOA antipatterns, we showed that SODA-W can specify and detect different Web services-specific antipatterns. We analysed more than 120 Web services and showed the accuracy of SODA-W with an average precision of more than 75% and recall of 100%.

In future work, we plan to enhance our approach to support other SOA styles, in particular REST services that follow different principles and standards for service design and consumption. Furthermore, we plan to conduct additional experiments with more Web services and antipatterns.

Acknowledgment The authors are thankful to Ons Mlouki for initiating the study. This study is supported by NSERC and FRQNT research grants.

References

1. Chambers, J., Cleveland, W., Tukey, P., Kleiner, B.: Graphical Methods for Data Analysis. Wadsworth International (1983)
2. Consel, C., Marlet, R.: Architecturing Software Using A Methodology for Language Development. Lecture Notes in Computer Science 1490, 170–194 (September 1998)
3. Coscia, J.A.L.O., Crasso, M., Mateos, C., Zunino, A.: Estimating Web Service Interface Quality Through Conventional Object-oriented Metrics. CLEI Electronic Journal 16 (April 2013)

4. Dudney, B., Asbury, S., Krozak, J.K., Wittkopf, K.: J2EE AntiPatterns. John Wiley & Sons Inc (August 2003)
5. Edwards, M.: Service Component Architecture (SCA). OASIS, USA (April 2011), `http://oasis-opencsa.org/sca`
6. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR (August 2005)
7. Evdemon, J.: Principles of Service Design: Service Patterns and Anti-Patterns (August 2005), `Online:msdn.microsoft.com/en-us/library/ms954638.aspx`
8. Frakes, W.B., Baeza-Yates, R.A.: Information Retrieval: Data Structures & Algorithms. Prentice-Hall (1992)
9. Heß, A., Johnston, E., Kushmerick, N.: ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services. In: In Proceedings of International Semantic Web Conference (2004)
10. 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)
11. Král, J., Žemlička, M.: Crucial Service-Oriented Antipatterns. vol. 2, pp. 160–171. International Academy, Research and Industry Association (IARIA) (2008)
12. 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)
13. Modi, T.: SOA Management: SOA Antipatterns. www.ebizq.net/topics/soa_management/features/7238.html (August 2006)
14. Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.G., Baudry, B., Jézéquel, J.M.: Specification and Detection of SOA Antipatterns. In: Service-Oriented Computing. Lecture Notes in Computer Science, vol. 7636, pp. 1–16. Springer Berlin Heidelberg (November 2012)
15. zur Muehlen, M., Nickerson, J.V., Swenson, K.D.: Developing Web Services Choreography Standards The Case of REST vs. SOAP. Decision Support Systems 40(1), 9–29 (2005)
16. 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)
17. Rodriguez, J.M., Crasso, M., Mateos, C., Zunino, A.: Best Practices for Describing, Consuming, and Discovering Web Services: A Comprehensive Toolset. Software: Practice and Experience 43(6), 613–639 (2013)
18. Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M.: Automatically Detecting Opportunities for Web Service Descriptions Improvement. vol. 341, pp. 139–150. Springer Berlin Heidelberg (2010)
19. Rotem-Gal-Oz, A., Bruno, E., Dahan, U.: SOA Patterns. Manning Publications Co. (2012)
20. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. Software: Practice and Experience 42(5), 559–583 (May 2012)
21. 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)
22. Sindhgatta, R., Sengupta, B., Ponnalagu, K.: Measuring the Quality of Service Oriented Design. vol. 5900, pp. 485–499. Springer Berlin Heidelberg (2009)