UniDoSA: The Unified Specification and Detection of Service Antipatterns

Francis Palma, *Student Member, IEEE,* Naouel Moha, *Member, IEEE,*
and Yann-Gaël Guéhéneuc, *Senior Member, IEEE*

Abstract—Service-based Systems (SBSs) are developed on top of diverse Service-Oriented Architecture (SOA) technologies or architectural styles. Like any other complex systems, SBSs face both functional and non-functional changes at the design or implementation-level. Such changes may degrade the design quality and quality of service (QoS) of the services in SBSs by introducing poor solutions—*service antipatterns*. The presence of service antipatterns in SBSs may hinder the future maintenance and evolution of SBSs. Assessing the quality of design and QoS of SBSs through the detection of service antipatterns may ease their maintenance and evolution. However, the current literature lacks a unified approach for modelling and evaluating the design of SBSs in term of design quality and QoS. To address this lack, this paper presents a meta-model unifying the three main service technologies: REST, SCA, and SOAP. Using the meta-model, it describes a unified approach, UniDoSA (Unified Specification and Detection of Service Antipatterns), supported by a framework, SOFA (Service Oriented Framework for Antipatterns), for modelling and evaluating the design quality and QoS of SBSs. We apply and validate UniDoSA on: (1) 18 RESTful APIs, (2) two SCA systems with more than 150 services, and (3) more than 120 SOAP Web services. With a high precision and recall, the detection results provide evidence of the presence of service antipatterns in SBSs, which calls for future studies of their impact on QoS.

Index Terms—Antipatterns, Service-based systems, REST, SCA, SOAP, Web services, Specification, Detection, Quality of service, Design, Software maintenance and evolution

1 Introduction

Service Oriented Architecture (SOA) [1]—a collection of principles and methods for designing and developing service-based systems (SBSs)—helps IT organisations to meet their business needs. SBSs developed using a SOA are composed of loosely-coupled, platform-independent, reusable functional units, *a.k.a.*, *services* [1]. SBSs are designed and implemented using a number of different architectural styles and technologies, typically REST [2], Service Component Architecture (SCA) [3], and SOAP Web services [4]. In this paper, we refer these technologies as *service technologies* and the systems developed using these technologies as SBSs [5].

Services and SBSs are not exempt of some common software-engineering challenges: maintenance and evolution. Maintenance and evolution take place when new or changed user requirements appear, typically due to: (1) *functional* changes, *i.e.*, changes at design and implementation levels and (2) *non-functional* changes, *i.e.*, changes in the execution contexts or in service-level agreements. All these changes may degrade the design quality (or implementation quality) and the quality of service (QoS) of SBSs and may cause the appearance of *service antipatterns*. Antipatterns are poor solutions to recurring design problems. The concept of *antipattern* was introduced by Koenig [6] as a way of documenting common mistakes in various phases of software development. Brown *et al.* [7] described an antipattern as a literary form that describes commonly occurring solution of a problem that may generate negative consequences. Naturally, antipatterns also exist in SBSs [8].

Multi Service and *Tiny Service* are examples of two typical service antipatterns [8]. *Multi Service* represents a service that implements a long list of operations varying in business abstractions. A service implemented as a *Multi Service* is not easily reusable and exhibits a low cohesion among its operations. Being overloaded by many different client requests, the *Multi Service* might become frequently inaccessible to its end-users. On the contrary, *Tiny Service*, a small service with very few operations, implements only a part of an abstraction, thus requiring several other tightly coupled services to complete an abstraction, increasing the design and development complexity. Researchers suggest that *Tiny Service* is the cause of failures in SBSs [9].

Antipatterns, for example *Multi Service*, may occur due to ignorance, *i.e.*, when engineers miss to identify proper business services when analysing requirements due to the absence of proper use cases [8]. This ignorance may lead to developing an interface for *Multi Service* with diverse/unrelated abstractions and operations, which must be detected and could be refactored. Such Multi Service interface may hinder maintenance and evolution in terms of cost and time. In the example of a *Tiny Service*, when engineers map the requirements and business services, every individual responsibility is mapped into a separate service [8]. This mapping results in introducing too many services (service

- *F. Palma is with the Screaming Power, Canada in collaboration with Ryerson University, Canada.*
  E-mail: francispalmaphd@gmail.com.
- *Y.-G. Guéhéneuc is with the Department of Computing and Software Engineering, Polytechnique Montréal, Canada.*
  E-mail: yann-gael.gueheneuc@polymtl.ca.
- *N. Moha is with the Department of Computer Science, University of Québec in Montréal, Canada.*
  E-mail: moha.naouel@uqam.ca.

interfaces) of small sizes into the system, increasing architectural and development complexity. Such mapping must also be identified automatically to facilitate maintenance and evolution for the SBSs in the long run. The detection of such service antipatterns is essential for the analysis of the design quality and QoS of SBSs.

However, this analysis is challenging because (1) service antipatterns do not have a formal specification and (2) each kind of SBSs technology, *i.e.*, REST, SCA, and SOAP, includes common and differing concepts with the others. With a few commonalities among them, the service technologies vary in their (1) building blocks, (2) composition styles, (3) development method, and (4) communication or client interaction styles. These variations pose some challenges to analyse them uniformly and effectively. They can also explain that the detection of service antipatterns in SBSs has not received much attention in the literature.

We identify three potential problems from the literature on the detection of service antipatterns in SBSs and propose solutions in this paper as follows:

- **No unified meta-model of various service technologies:** To detect service antipatterns independent of SBS technologies (*e.g.*, REST, SCA, and SOAP) uniformly, we need a unified meta-model to apply generic approaches to specify and detect service antipatterns effectively. This unified meta-model must provide concepts from the diverse technologies and their inter-related concepts. However, combining various technologies is challenging because, despite some commonalities, these technologies differ in architectural styles and on how their clients consume services.
  Solution: Having a **unified meta-model** is the first step before proposing a effective approach for the specification and detection of service antipatterns in service technologies.
- **No specification of service antipatterns:** To detect service antipatterns, we must specify them in a machine-processable yet human-understandable format. Without proper specifications, service antipatterns are ambiguous and cannot be detected automatically. The present service-antipatterns literature describes antipatterns textually and informally, which is hard to handle and use.
  Solution: Specifying service antipatterns using a **specification language** in an automatable format is the pre-requisite step for their automatic detection.
- **No dedicated unified approach and framework for the detection of service antipatterns:** SBSs operate in Internet-based dynamic environment. The execution contexts of SBSs, *i.e.*, scenarios based on choreographed services, and their dynamic nature, *i.e.*, physical availability and service-level agreements between services and clients, make the analysis of services and the detection of service antipatterns challenging. There are numerous contributions in the literature for the detection of object-oriented (OO) antipatterns, *e.g.*, [10], [11], [12], [13], [14], [15], [16], [17], [18]. Yet, there is no consolidated method and technique for such detection in SBSs.
  Solution: Providing a **unified approach based on a generic framework** facilitates the detection of service antipatterns in SBSs regardless of their underlying technologies and increases the reusability and extensibility of the framework.

In the literature, some progresses have been made in the domain of SOAP Web services [9], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], of RESTful APIs [31], [32], and of RESTful APIs for cloud services [33], [34]. In our previous works [5], [31], [35], we proposed technology-specific approaches for REST [31], SCA [5], and SOAP [35]. In this paper, to perform the detection of service antipatterns uniformly, we combine those technology-specific contributions by (1) proposing a unified meta-model that unifies the concepts in the technologies to specify service antipatterns effectively and (2) providing an integrated and extensible framework capable of detecting service antipatterns in various SBSs.

Thus, we present a novel and innovative approach, UniDoSA (**Uni**fied Specification and **D**etection **o**f **S**ervice **A**ntipatterns), relying on a framework, SOFA (Service Oriented Framework for Antipatterns), for the unified specification and automatic detection of service antipatterns in SBSs. SOFA facilitates the static and dynamic analyses of SBSs where the static analysis concerns quantifying design-related structural properties and the dynamic analysis refers to quantifying runtime properties while executing an SBS. Our proposed UniDoSA approach relies upon a domain specific language (DSL) to specify service antipatterns in terms of metrics, both static and dynamic, at higher-level of abstraction. The DSL is based on a unified meta-model defined after a thorough domain analysis of service antipatterns from the literature. With the help of the DSL and SOFA, we automatically generate detection algorithms from the specifications of service antipatterns, later, we apply them on a target SBS.

We show the usefulness of UniDoSA by defining rules for 12 service antipatterns from REST, SCA, and SOAP and by performing their detection. We validate the detection results in terms of precision, recall, and F_1-measure on: (1) 18 well-known REST APIs, including Facebook, Twitter, Dropbox, and YouTube, (2) *FraSCAti*, the largest open-source SCA system with 130 services and *Home-Automation* [36], a demo SCA application with 13 services, and (3) more than 120 SOAP services collected from the Web services search engine `programmableweb.com`. We show that UniDoSA allows the specification and detection of a representative set of service antipatterns with a high average precision and recall, *e.g.*, more than 75%.

Thus, with this paper, we contribute to the literature on antipatterns and on SOA with:

- A **unified meta-model** combining different service technologies and architectural styles showing the differences and commonalities among them;
- Based on the unified meta-model, a **unified DSL** to specify service antipatterns regardless of service technologies with higher-level abstractions;
- A **unified approach, UniDoSA**, relying the unified meta-model and DSL, for the specification and detection of service antipatterns in SBS technologies;

- An **extensive validation** of UniDoSA on 18 well-known RESTful APIs, more than 150 services from two SCA system, and more than 120 SOAP services;
- An **online prototype tool**, WEBRESTPAD, for the detection of service antipatterns, in particular for RESTful APIs that are more popular and widely used by service providers.

The remainder of this paper is organised as follows. Section 2 briefly discusses the background of this paper, comparing the three SBS technologies. Section 3 presents the unified meta-model for the three service technologies. Section 4 shows that the unified UniDoSA approach can be used to efficiently and effectively specify and detect different service antipatterns relying on a unified framework, SOFA. Section 5 presents a detailed example where we apply our UniDoSA approach using a common service antipattern, *Multi Service*. Section 6 discusses the experiments and the results we obtained. Section 7 highlights relevant works in the literature through a literature survey and shows how this paper fills in existing gaps. Finally, Section 8 concludes and sketches future work.

2 BACKGROUND

In this section, we briefly introduce the three service technologies considered in this paper: REpresentational State Transfer (REST) [2], Service Component Architecture (SCA) [3], and SOAP Web services [4]. We also show a detailed comparison among them. Later, we perform a literature review to identify contributions on antipatterns detection pertaining to these service technologies and to identify gaps in the current literature.

2.1 Service Technologies

We consider the following three service technologies because, as of today, they are dominant in terms of acceptance in the industry due to their simplicity in publishing and consuming services over the Web [37].

2.1.1 REpresentational State Transfer (REST)

REST allows resource-centric remote services [2]. Unlike SOAP services below, which operate using customised operations, REST services use standard HTTP operations, *e.g.*, GET, POST, PUT, and DELETE, to access and manipulate resources, thus increasing data transport efficiency and reducing data handling complexity. This efficiency of REST comes from its light-weight design and simple usage scheme. The unique characteristics of REST architectures are: (1) the explicit use of HTTP methods, (2) the statelessness and cacheability, thus scalability, (3) the exposure of directory-like URIs (Uniform Resource Identifiers), and (4) the ability to transfer data in many Web formats, including XML and JSON (JavaScript Object Notation).

2.1.2 Service Component Architecture (SCA)

SCA is a software technology that provides a model to compose applications on top of SOA design principles [3]. The composition, *a.k.a.*, *SCA composite*, is described using a standard XML-based language, SCDL (Service Component Definition Language) where a set of related *SCA components* are orchestrated. The components provide the actual desired business functionalities in the form of *services*. SCA defines a technology-agnostic model for composing diverse interface definition languages (WSDL or Java), implementation languages and frameworks (Java, BPEL, C/C++, Spring, or OSGi), and bindings (SOAP, JMS, or REST).

2.1.3 SOAP Web Services

SOAP Web services rely on the XML-based messaging protocol SOAP (Simple Object Access Protocol) [4] and operate using customised operations. The communications between clients and SOAP services are based on standards: (1) XML (eXtensible Markup Language) as the service data format, (2) HTTP as the transport protocol, (3) SOAP as the reliable and secured messaging protocol, (4) UDDI (Universal Description, Discovery, and Integration) as the service discovery mechanism, and (5) WSDL (Web Services Description Language) as the formal service contract.

2.2 Comparison of the Service Technologies

We now compare the three service technologies. Differences at the architectural, design, and implementation-level exist among the service technologies, as summarised in Table 1. An extensive review of the literature help us to identify and classify the various technology-specific properties highlighted in Table 1, which we must consider when analysing various service-based systems.

The next two sections discuss the differences among the three service technologies in two aspects: their core design elements and service consumption styles.

2.2.1 Core Design Elements

A first major difference among the three service technologies is in their core design elements.

REST relies on resources that can be anything from one single piece of data (*e.g.*, a name, a salary) to a complete file (*e.g.*, JPEG or PDF). Resources are identified using URIs (Uniform Resource Identifiers) and are accessible via standard HTTP methods. REST resources can also be a collection of other resources, *i.e.*, composite resources.

SCA relies on *component* as its building block. A component provides a specific service and implements at least one interface [3]. A collection of related components are specified in a SCA composite and these components work together to achieve a higher-level business goal.

A *service* relying on the SOAP protocol is operation-centric and exposes an arbitrary set of customised operations. Clients can search a service directory for their desired customised interfaces. One use of such services is by orchestrating them using a well-defined structured language, like BPEL4WS [38].

2.2.2 Service Consumption Styles

Differences among the three service technologies exist regarding their use, *i.e.,* the consumption of their services. The diverse service consumption styles may introduce diverse technology-specific antipatterns, which we summarise in the following.

TABLE 1: Non-trivial Architectural Differences among REST, SCA, and SOAP technologies.

Comparison Criteria	SOAP	SCA	REST
Cacheability	no	no	cacheable
Contract design	contract first/last	contract last	contract-less
Dynamic configuration management	no	yes	no
Dynamic deployment	no	yes	no
Error handling	no	built-in	built-in
Message encoding	yes	yes	no
Messaging support	yes	within domain vendor-specific	no
Policy	WS-Policy	SCA policy framework	no standard
Operations invoking protocol	***SOAP***	***SOAP, JMS, RMI***	***HTTP***
Reliability	WS-Reliability	non-standard	no
Representation of information	***XML-standard***	***XML-standard***	***JSON, XML, MIME, so on***
Security	WS-Security	SCA security policy	HTTP, SSL
Standards based	yes	yes	no
Statelessness	mostly stateful	by default stateless	completely stateless
Transactions	WS-AtomicTransaction	WS-AtomicTransaction	no standard
Transport protocol	HTTP, TCP, SMTP, JMS	HTTP, TCP, SMTP, JMS	HTTP
Verbosity	more	more	less
Service composition	***WS-BPEL***	***SCDL***	***mashups***
Service/resource identification	***WS-Addressing***	***no***	***URI***
Core design elements	***service***	***component***	***resource***
Focus	accessing named operations	accessing components as service units	accessing named resources
Human intelligible payload	no	no	yes
Hypermedia/hyperlinking	***no support***	***no support***	***natural support***
Interface	different interfaces for services	different interfaces for components	uniform interface for resources
Service discovery	UDDI registries	not applicable	no standard
Service invocation	***through calling RPC method***	***through calling RPC method***	***via URL path***
Standardised interface definition	***Web Services Description Language***	***Service Component Definition Language***	***no***
Interface exposure	public	neither	neither
Method callability	exposed as remotely callable operation	no	no
Specification	JAX-WS	SCA-J	JAX-RS
Written documentation	***no dependency***	***no dependency***	***highly dependent***

1) REST has no standardised contract or specification. SOAP services have public, discoverable WSDL-based contracts, *a.k.a.*, service interfaces. SCA systems have SCDL-based specifications that are private and non-discoverable. Generating or writing those service contracts and specifications must follow standardised conventions and best practices to allow their understandability and re-usability.
   Summary*: Depending on the service technology, the appearance of service antipatterns related to service contracts might vary.*
2) The SOAP clients invoke services relying only on the SOAP protocol. SCA also executes its component services relying on SOAP, JMS, or REST protocols when various SCA composites are not within a single machine. As for REST, the service invocation is completely dependent on HTTP and relies on client requests based on resources URIs.
   Summary*: Antipatterns in REST depend on how well the HTTP client requests are formed following best practices for REST described in the literature from the client side and how well the HTTP responses are designed from the server side. Thus, depending on the technologies the types of antipatterns may vary.*
3) The response data and exchanged messages are available only in XML form for SOAP services, in XML or SDO (Service Data Objects) for SCA systems, and in any Web formats, like JSON, XML, MIME, YAML, or PDF, for REST.
   Summary*: For SOAP and SCA services, having only one data representation is not a bad practice whereas REST must facilitate multiple representations of the same resource and having only one representation is considered a bad practice in REST.*

2.3 Overview of Service Antipatterns

Designers or developers may follow technology and developer-neutral bad practices while designing or developing SBSs using different service technologies. Thus, service antipatterns exist in the different service technologies, REST, SCA, and SOAP.

Figure 1 shows a Venn diagram relating 12 service antipatterns found in REST, SCA, and SOAP services. Table 2 presents the concise textual descriptions of the 12 services antipatterns of interest in this paper. We identify and relate these 12 service antipatterns from the literature as defined in [8], [9], [39], [40], [41], [42], [43], [44]. It is important to mention that to the best of our knowledge, all the research activities reported in Section 7 deal with the detection of technology-specific service antipatterns only.

In Figure 1, some antipatterns are specific while others are common to the different service technologies. Antipatterns may be of types inter-service (involve other services in a SBS) or intra-service (not depending or impacting other services directly) and may require static, dynamic, or hybrid analyses of services to be detected.

REST APIs clients do not need to know concrete interfaces to the services that they consume and only send requests using well-known HTTP methods. Thus, REST

TABLE 2: List of 12 Antipatterns in REST, SCA, and SOAP Technologies.

Antipattern(s) in SCA, REST, and SOAP
Ambiguous Name 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***. *Ambiguous names* are not ***semantically*** and ***syntactically*** sound and impact the ***discoverability*** and the ***re-usability*** of a service [39].
Bloated Service is an antipattern related to service implementation where services or resources in SOA become 'blobs' with ***one large interface*** and–or ***lots of parameters***. *Bloated Service* performs ***heterogeneous operations*** with ***low cohesion*** among them. It results in a system with less maintainability, testability, and reusability. It requires the consumers to be aware of many details (*i.e.*, parameters) to invoke or customize them [40].
Antipattern(s) in SCA and SOAP
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 service aggregates too many methods 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 it is ***overloaded***, which may also induce a ***high response time*** [8].
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 [8].
Antipattern(s) in SCA and REST
Nobody Home (*Deprecated Resource* in REST) corresponds to a service (or resource), defined but actually never used by clients. Thus, the ***methods*** from this service (or related to this resource) are ***never invoked***, even though it may be ***coupled*** to other services. Yet, it still requires deployment and management (or appears in the API documentation), despite of its non-usage [41].
Antipattern(s) in SOAP and REST
In *CRUDy Interface* antipattern, the design encourages services to adopt 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*** [42].
In REST, the URIs with ***CRUDy verbs*** (*e.g.*, *create*, *read*, *update* or *delete*) or their ***synonyms*** can be confusing for API clients, *i.e.*, conceptually, it overloads the ***HTTP methods***, and thus, introduce *CRUDy URI* antipattern. Using CRUDy terms in a URI prohibits users to use appropriate HTTP methods applicable to a certain context [43].
Antipattern(s) only in SCA
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 [9].
Antipattern(s) only in SOAP
Chatty Web Service 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*** [8].
Antipattern(s) only in REST
Forgetting Hypermedia: The ***lack*** of ***hypermedia***, *i.e.*, ***not linking resources***, hinders the state transition for REST applications. One possible indication of this antipattern is the ***absence*** of URL ***links*** in the ***resource representation***, which typically restricts clients to follow the links, *i.e.*, limits the dynamic communication between clients and servers [44].
Ignoring MIME Types: The server should represent ***resources*** in various formats, *e.g.*, ***xml, json, pdf***, etc., which may allow clients, developed in diverse languages, a more flexible service consumption. However, the server side ***developers*** often intend to have a ***single representation*** of resources or rely on their ***own formats***, which limits the resource (or service) ***accessibility*** and ***re-usability*** [44].

antipatterns are defined in the literature focusing on best practices of making client requests and sending server responses. For example, *Forgetting Hypermedia* and *Ignoring MIME Types* antipatterns indicate that the HATEOAS principle is not enforced and that content-negotiation mechanism are lacking [44].

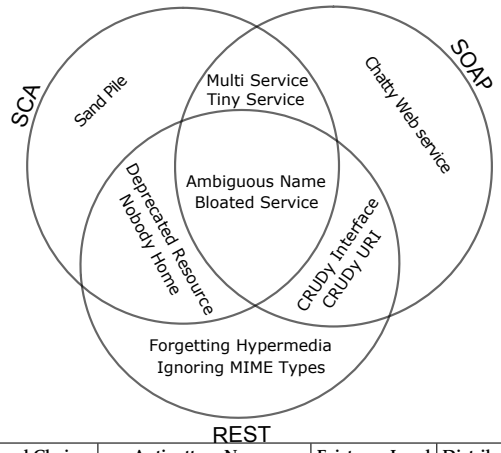
The antipatterns in SCA and SOAP services are described in the literature based on various criteria related to services design and implementation and their runtime behavior (*e.g.*, *Multi Service*, *Tiny Service*, and *Chatty Web service*) because in SCA and SOAP services, clients consume services by invoking operations defined in their service interfaces and the runtime behavior of services depend on how well the services are designed and implemented.

The detection algorithms for common antipatterns, shown in Figure 1, are similar across service technologies but require a unified model of these service technologies. The detection algorithms for technology-specific antipatterns will of course vary. Figure 1 shows a comparison among antipatterns in REST, SCA, and SOAP services. Antipatterns in SOAP services are defined at the service interface-level whereas antipatterns in SCA are at component-level. The *components* in SCA are at a higher level of granularity than the *services* in SOAP. Moreover, the antipatterns in REST applicable at the resources level.

In this paper, we analyse SOAP services as individual entities, *i.e.*, not as composition of services. Therefore, none of the discussed antipatterns for SOAP services are at the service composition-level. Nevertheless, there are antipatterns that spread across several SOAP services, *e.g.*, *Stovepipe Service* or *Single-Schema Dream* [8], which we will analyse in future work because analysing a set of services requires at least one execution scenario defined using a process language [38].

In summary, in this section, we compare different service technologies and identify a list of non-trivial differences that describe the technology-specific characteristics or properties. These different characteristics yield different antipatterns from one technology to another. We show the differences and commonalities among antipatterns in various technologies in Section 2.3 and conclude that:

"*Despite the presence of some common antipatterns across diverse service technologies, there exist also antipatterns that are technology-specific, and, therefore, their specification and detection may differ, which poses the challenge to have a unified and technology-independent approach for the detection of service antipatterns in different service technologies*."



Architectural Choices	Antipattern Names	Existence Level	Distribution Level
SCA ∩ REST ∩ SOAP	Ambiguous Name [39]	Interface	Intra-service
	Bloated Service [40]		
SCA ∩ SOAP	Multi Service [8]		
	Tiny Service [8]		
SCA ∩ REST	Nobody Home [41]	Component	
	Deprecated Resources	Resource	Intra-resource
REST ∩ SOAP	CRUDy Interface [42]	Interface	Intra-service
	CRUDy URI [43]	Resource	Intra-resource
SCA	Sand Pile [9]	Composition	Inter-service
SOAP	Chatty Web Service [8]	Interface	Intra-service
REST	Forgetting Hypermedia [44]	Resource	Intra-resource
	Ignoring MIME Types [44]		

Fig. 1: Set Relation among Service Antipatterns in REST, SCA, and SOAP technologies.

3 Unified Meta-model

We propose a unified meta-model for service technologies. To build this unified meta-model, we build, study, and combine the meta-models of the three technologies, as shown in Figure 2, depicting their commonalities and differences at their design and implementation levels. We benefit from the proposed unified meta-model to define a unified domain-specific language for specifying the service antipatterns at a higher level of abstraction for their representation. Section 4 details how we define and use such a unified domain language for the specification and detection of service antipatterns in the three technologies.

We introduce the unified meta-model for REST, SCA, and SOAP services in Figure 2. Individual meta-models of REST, SCA, and SOAP services can be found in previous works: [45], [46], and [47], respectively. In Figure 2, we present the simplified versions of their meta-models by hiding optional details not related to services design and implementation.

To develop our unified meta-model, we follow an iterative and incremental process. We start with the multiple technology-specific models, and, as the first step, identify their core elements, *i.e.*, the basic building blocks for each technology, including Component for SCA, REST service for REST, and SOAP service for SOAP. We then consider other related elements that are mandatory for the existence of these core elements. For example, an SCA component must define at least one service (*i.e.*, what it is doing) and a REST service must have at least one resource (*i.e.*, what it operates on). Thus, incrementally, we develop a 'complete' meta-model through iterations, *i.e.*, a small segment with few model elements at a time. We come to a stop only when we all agree on a model segment after each iteration. The process of building our unified meta-model is inspired from the Unified Software Development Process [48].

Our proposed unified meta-model has six parts, each representing a distinct area within the model. The elements in blue on the left of the meta-model represents concepts specific to REST while the elements in green represents SCA-specific concepts in the model. The central-top elements of the model in red corresponds to the concepts specific to SOAP services. The core part of the proposed meta-model, *i.e.*, elements in black, includes the concepts common in all three technologies studied in this paper and is marked as REST ∩ SOAP ∩ SCA. Finally, The elements in green on the right of the meta-model represents concepts specific to SCA. In the following we discuss some interesting parts of the proposed meta-model.

3.1 REST ∩ SOAP ∩ SCA

As shown in Figure 3, the core part of the proposed unified meta-model includes the concepts common in all three SBS technologies. All the three SBS technologies implement at least one `SecurityMechanism` to make service consumption secured. For example, REST may rely on OAuth whereas SCA may rely on SCA Security Policy and SOAP uses WS-Security. Moreover, all these technologies use a specific `Transport` protocol. The HTTP protocol, for example, can be used both by REST and SOAP. Each technology shares the notion of `Service` through which operations or methods become also integral part of these three technologies. REST, SCA, and SOAP technologies also share the notion of service `Endpoint`. Finally, all these three SBS technologies rely on a specification mechanism that either exposes their capabilities (*e.g.*, HTML/PDF for REST or WSDL for SOAP) or their configuration details (*e.g.*, SCDL for SCA).

3.2 REST

Among REST-specific concepts in the proposed meta-model are the `Resource`, `RESTService`, `Method`, `Request`, and `Response` concepts, which are worth to discuss in details. The `RESTService` is based on URI conventions and defined as a collection of multiple `Resources` that can be accessed via a `baseURI`, *a.k.a.*, an `Endpoint`. To access a `Resource`, at least one `Method` of type `HTTPMethods` must be defined by a REST developer or offered by one REST service. HTTP methods (*e.g.*, GET, POST, PUT or DELETE) are used by client to the REST services to request a `Resource`. In addition, a REST `Request` and `Response` may include a `Body`. Each `Resource` has a `Representation` of type `MediaTypes`.

3.3 SCA

This part of the meta-model includes concepts specific only to SCA, including the `Reference`, `Component`, `Wire`, `Property`, `Composite`, and others concepts. The design and implementation of SCA systems require these concepts, which are not found in REST and SOAP meta-models. In

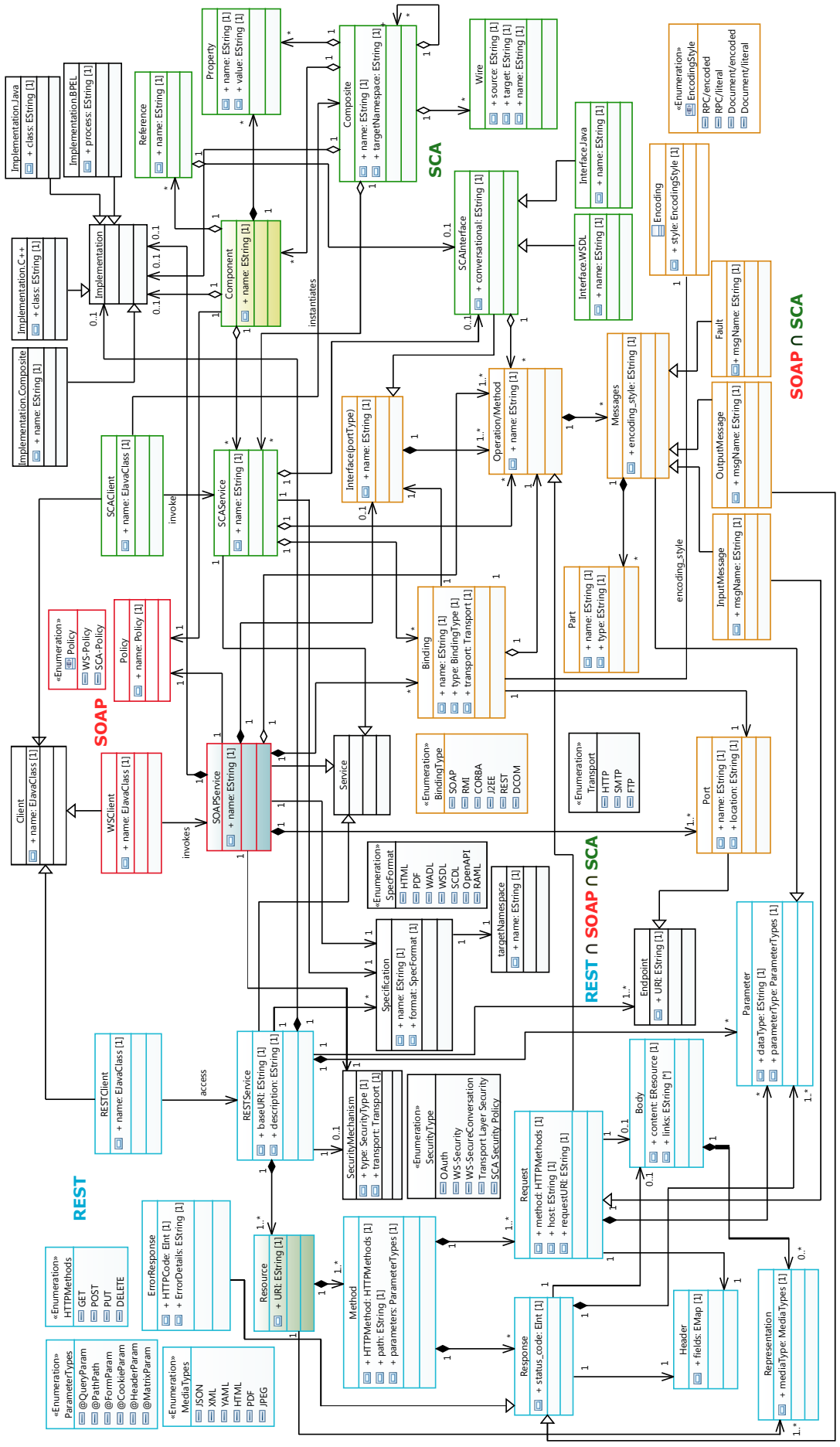


Fig. 2: Unified Meta-model for REST, SCA, and SOAP Technologies.

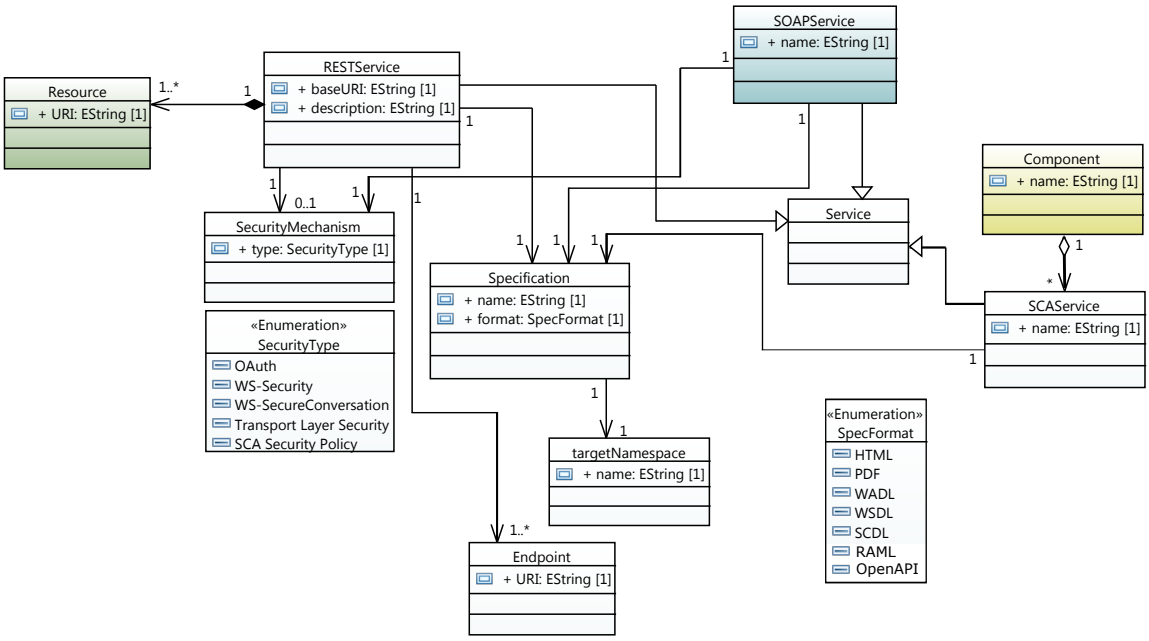


Fig. 3: Elements Common in REST, SCA, and SOAP Technologies (REST ∩ SCA ∩ SOAP).

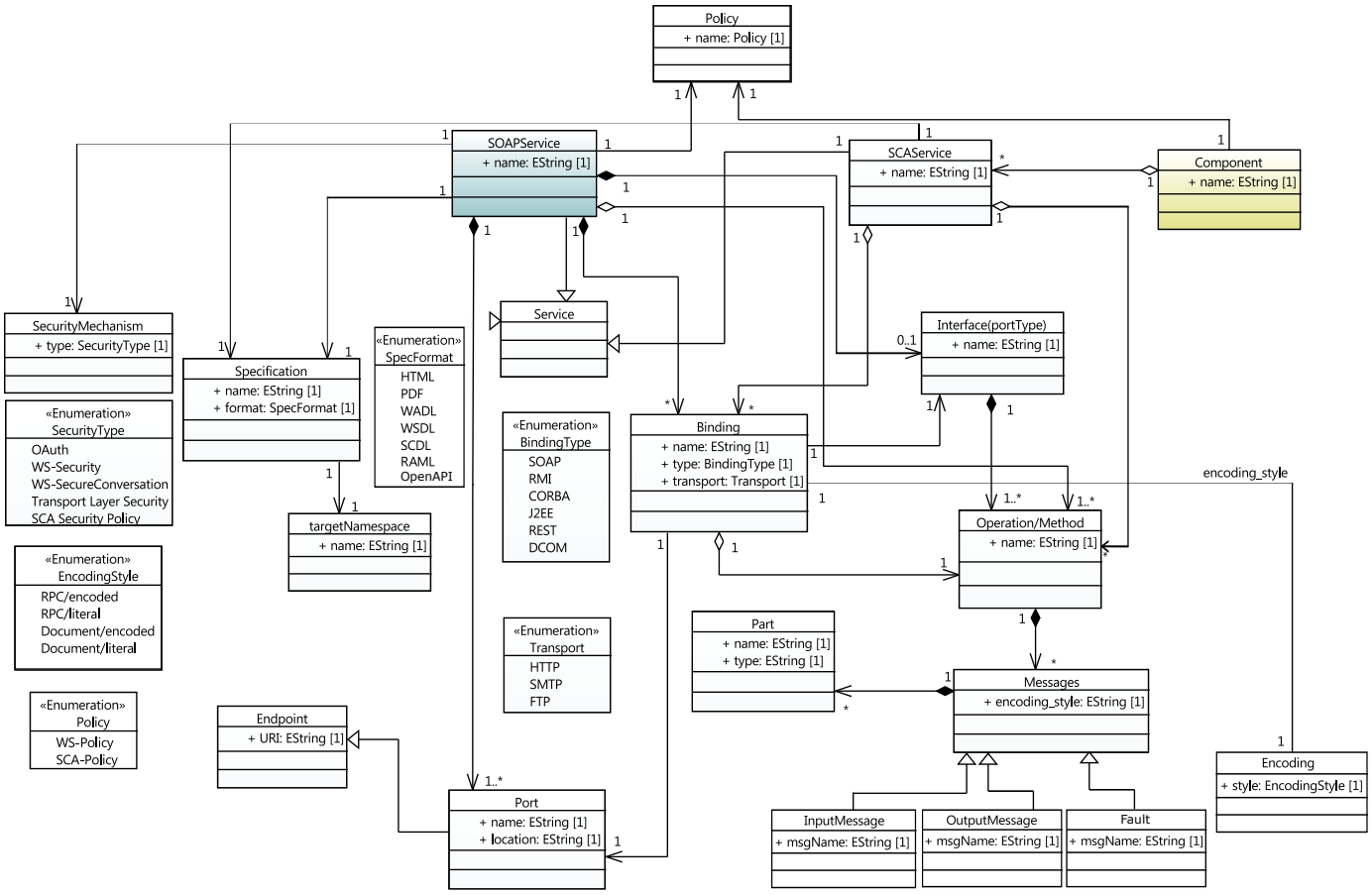


Fig. 4: Elements Common in SCA and SOAP Technologies (SCA ∩ SOAP).

SCA, a `Component` provides at least one `Service` and several `Components` may reside within a `Composite`. Multiple `Composites` may be connected via `Wires` and `Components`, which might be dynamically reconfigured via various `Property` values.

3.4 SOAP

The concept that only belongs to SOAP Web services is the `Policy` concept. The rest of the concepts needed to describe SOAP-based SBSs are either common with REST or SCA. For example, SOAP shares with SCA the concepts of `Interface`, `Binding`, `Security`, `Policy`, and `Operation/Method`. SOAP also shares with REST the concepts of `Endpoint` and `Transport`. In general, SOAP shares more commonality with REST and SCA, unlike the few common concepts that exist in REST and SCA.

3.5 REST ∩ SOAP

This part of the meta-model represents the intersection between REST and SOAP meta-models. A set of arguments, *a.k.a.*, `Parameter`, must be specified to make a HTTP `Request`. The most recent state of the requested resource is returned as a `Response` message that must have a `Representation`, which is generally in a globally accepted representation format rather than MIME, *e.g.*, XML, JSON, or PDF for REST, and only in XML for SOAP. Thus, in the proposed unified meta-model, the `Representation` is defined as an abstract entity, which is generalised in various data representation formats.

3.6 SOAP ∩ SCA

Figure 4 shows the concepts common in SOAP and SCA services where a `Service` implements an `Interface` and each service has at least one `Method/Operation` defined and at least one `Binding` specified binding to a service. The concepts `Interface/PortType`, `Service`, `Method/Operation`, and `Binding` are common in SOAP and SCA meta-models as well as `Policy` and `SecurityMechanism` concepts.

3.7 Discussion

This first unified meta-model, in Figure 2, of the three most popular service technologies brings the following benefits:

1) It is helpful in specifying and identifying rather than analysing service antipatterns in SBSs uniformly, as much independent of technologies as conceptually possible, by providing a common language to specify service antipatterns at a higher-level representation abstraction and without ambiguity, *i.e.*, no multiple semantics of the same antipattern concept;
2) It clearly separates the technology-specific concepts and relates them where applicable, which is helpful in understanding the commonalities and differences among various service technologies;
3) The specification of service antipatterns uniformly allows building a unified approach for their detection in service-based systems regardless of their underlying technologies.

Our unified meta-model is applicable to model any SBSs developed with REST, SCA, or SOAP services. It is, in addition, extensible for new technologies by integrating their shared concepts, for example, architectural components and communication styles.

For example, if it is required to add the WCF (Windows Communication Foundation) service technology in our unified meta-model, the identification of basic concepts in WCF is the first step. A WCF service exposes a collection of endpoints where each endpoint is a portal to communicate with external services. A service endpoint has an address (a network address), a binding (how the end-point communicates with other services including transport protocol (*e.g.*, TCP, HTTP), encoding (*e.g.*, text or binary), and security requirements (*e.g.*, SSL or SOAP message security)), and a contract (what the endpoint communicates), and is a collection of messages organised in operations. Moreover, an endpoint address is formed using a URI and an identity. A binding, for example, has a name and a namespace.

These concepts above are from WCF technology and are closely related to SOAP-based Web services. Therefore, while comparing with SOAP, it is expected that the common elements between WCF and SOAP would be relatively numerous when compared to other technologies.

Then, to accommodate WCF, we would start by adding the most basic element, WCF service, in the model and incrementally add or map other elements depending which pre-existing model elements match with the concepts in the new technology we want to add. In summary, after a thorough domain analysis, it is achievable only by domain experts with the knowledge of existing unified meta-model.

4 Proposed Unified Approach

Relying on our unified meta-model, we now present a unified approach for the specification and detection of service antipatterns in SBSs written in either one of the three service technologies REST, SCA, and SOAP.

Figure 5 shows our proposed unified approach, UniDoSA (Unified Specification and Detection of Service Antipatterns), for specifying and detecting service antipatterns in SBSs. Starting with the textual descriptions of service antipatterns and ending with the validation of the detected occurrences of these antipatterns, the four main steps of UniDoSA include:

Step 1: Specification. This step includes performing a thorough domain analysis by studying the definitions and textual descriptions of antipatterns from the literature to identify the relevant static and dynamic properties of the elements of the unified meta-model to specify them. The identified properties represent (1) measurable attributes of the elements of the proposed meta-model and (2) the relations among elements. Thus, we use the elements of the unified meta-model as the vocabulary to define a common domain specific language (DSL) to formalise antipatterns with rule cards. Rule cards are representations of antipatterns at a higher-level of representation abstraction, which are both machine processable and human readable. Figure 10 shows examples of rule card for *Multi Service* and *Tiny Service*. Figure 12 shows the rule card of *Forgetting Hypermedia* REST antipattern. For REST antipatterns, rule cards are applied on the HTTP requests and responses.

Step 2: Generation. From the rule cards in the previous step, we generate automatically their detection algorithms using a *template*-based technique. We define templates of detection

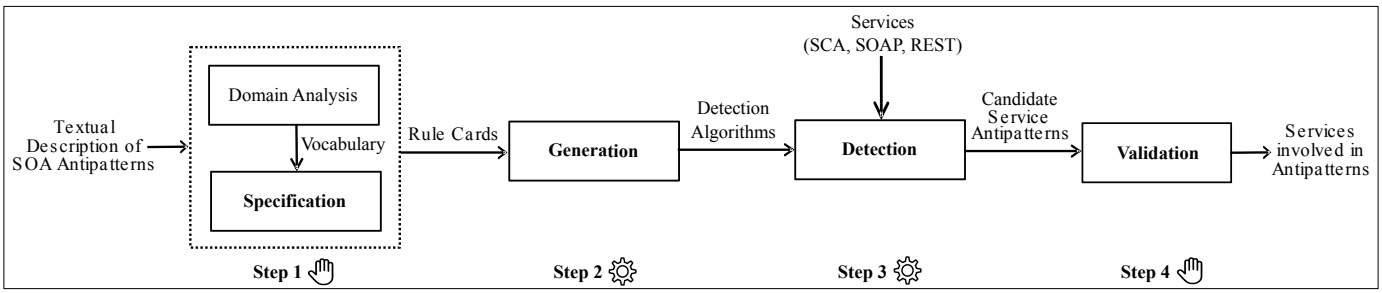


Fig. 5: UniDoSA Approach for the Specification and Detection of Service Antipatterns in SBSs.

algorithms with well-defined *tags*, which we replace by concrete values at generation-time of the detection algorithm. This step results in detection algorithms (*i.e.*, Java programs) for each service antipattern that we have specified and that we want to detect. In fact, manually implementing Java programs would be more complicated and at a much lower-level of abstraction. Thus, we rely on a technique for the automatic generation of the detection algorithms.

Step 3: Detection. For the detection of antipatterns, we introduce the framework SOFA (Service Oriented Framework for Antipatterns). The computations of all static and dynamic metrics or properties are performed by SOFA. Our SOFA framework also assists in semantic analysis of services interfaces. In this step, we apply the detection algorithms automatically generated in the previous step on different SBSs and report candidate service antipatterns as the occurrences of service antipatterns. Moreover, for REST services, SOFA provides the means to concretely send HTTP requests and to apply automatically the detection algorithms on both HTTP requests and responses.

Step 4: Validation. In this step, we rely on independent engineers and external developers to manually validate the suspected services, *i.e.*, the occurrences of service antipatterns, obtained in the previous step to verify that those services indeed suffer from the antipatterns as described in the literature and as listed in Table 2 in form and semantics, the latter being only possibly validated by developers.

Our UniDoSA approach is designed to work for any number of antipatterns regardless of technology, in particular, currently, for the SOAP, SCA, and REST service technologies.

The next two sections describe the specification of service antipatterns and the generation of their detection algorithms in further details. We describe SOFA in Section 4.3. We discuss the validation of the occurrences in Section 6.

4.1 Specification of Service Antipatterns

As the first step towards the specification of service antipatterns from their textual, informal descriptions in the literature, we perform a thorough domain analysis of service antipatterns by studying their definitions and textual descriptions [9], [8], [19] and in online resources and articles [40], [39], [42], [49], [41]. The process of domain analysis involves identifying, capturing, and organising reusable information for using them in software development [50]. The domain analysis allows us to identify properties relevant to service antipatterns, including static properties related to their design (*e.g.*, cohesion and coupling) and dynamic properties, such as QoS properties (*e.g.*, response time, availability, or throughput) and to relate these properties to the elements of the unified meta-model describing the three service technologies.

Static properties are properties that apply to the static descriptions of SBSs, such as WSDL files for SOAP services and SCDL files for SCA services (under the `Specification` element in the meta-model) whereas dynamic properties are related to the dynamic behavior or nature of SBSs as observed during their execution. Our proposed unified meta-model (see Figure 2) represents the elements necessary to compute static properties.

Not all service antipatterns require the computation of dynamic properties but it is possible using our meta-model. The dynamic or behavioral properties of services in the meta-model are related to the `SCAService`, `SOAPService`, and `RESTService` elements. Dynamic properties for each of these elements include, for example, response time, availability, and throughput. These dynamic properties can only be measured by concretely invoking the services. Table 3 shows the elements in our meta-model and what we measure on them. The last column in Table 3 shows how we identify the measurement of elements. This identification also works as the basis of our DSL. After analysing the descriptions of the service antipatterns (see Table 2), we identify from our meta-model the elements on which to perform the measurement and how we can perform the measures while we define the rule cards for antipatterns.

We compute the dynamic properties by relying on an external API called SoapUI[1]. To obtain the best possible estimate, for each service, we randomly select five operations (with minimal or no parameters), invoke them ten times, and take the average of the measures. However, the measurement of dynamic properties may vary depending on the Internet traffic. To address the traffic issue and not to bias the detection, we rely on taking the average of several invocations, *e.g.*, ten invocations of the same service.

For example, in our UniDoSA approach, the *Ambiguous Name*, *CRUDy URI*, and *CRUDy Interface* service antipatterns require only the static analysis where as *Multi Service* or *Chatty Web Service* antipattern requires both the static and dynamic analyses for its detection.

Table 2 lists antipatterns for REST, SCA, and SOAP services selected from the literature and which are commonly found in SBSs. We highlight their various relevant

1. www.soapui.org/apidocs/overview-summary.html

TABLE 3: The Elements in our Meta-model and What We Measure on Them.

Element Name	Domain(s)	What We Observe or Measure on the Element	Identified by
Body	REST	Count of links in response body	TLB
		Count of verbs in request body	VRB
Header	REST	Presence of authentication cookie	AC
		Presence of client cookie	CC
		Presence of client caching value	CCV
		Presence of entity tag	ET
		Presence of header link	HL
		Presence of location field	LF
		Presence of server cookie	SC
		Presence of server caching value	SCV
Method	REST	The http method used	HM
Representation	REST	The resource representation format	RRF
Request	REST	Presence of 'action' keywords	AK
		Presence of verbs in request URI	VRU
Service	REST, SOAP, SCA	The availability of a service	A
		The coupling of a service	CPL
		The cohesion among operations in a service	COH
		Total number of method invocations	NMI
		Total number of transitive methods invoked	NTMI
		The response time of a service	RT
Interface(portType), Operation/Method, Messages	SOAP, SCA	The average length of signatures	ALS
		The ratio of general terms in signatures	RGTS
Interface(portType)	SOAP, SCA	The average ratio of identical port-types	ARIP
		Total number of interfaces in a component	NI
		Total number of operations in port-types	NOPT
		The total number of port-types in an interface	NPT
Operation/Method	SOAP, SCA	The average ratio of identical operations	ARIO
		The average number of accessor operations	ANAO
		The average number of identical operations	ANIO
		The average number of parameters in operations	ANP
		The average number of primitive type parameters	ANPT
		The total number of operations declared	NOD
		The total number of verbs in operation signatures	NVOS
		The total number of CRUD operations	NCO
		The total number of utility methods	NUM
Messages	SOAP, SCA	The average ratio of identical messages	ARIM
		The total number of verbs in message signatures	NVMS
Component	SCA	The total number of services encapsulated	NSE
		The total number of parameters in a component	TNP
Reference	SCA	The total number of incoming references	NIR
		The total number of outgoing references	NOR

properties unique to antipatterns that are measurable in *bold-italic*. We use these properties and the elements of the unified meta-model as the vocabulary to define a DSL, in the form of a rule-based language, for specifying service antipatterns. The DSL offers software engineers higher-level domain-related abstractions and variability points (*e.g.*, five-point Likert scale from `VERY_LOW` to `VERY_HIGH`) to specify service antipatterns based on their judgments, experiences, and contexts.

We manually identify and organise relevant domain concepts and properties essential for specifying service antipatterns via rule cards at a high-level of abstraction using a DSL. With our domain analysis, specifications of service antipatterns are made in a consistent high-level abstraction and capture all relevant domain expertise. Thus, for the domain experts, it becomes easy to understand and modify the specifications without prior knowledge of the underlying detection framework.

The DSL provides the engineers with high-level, domain-related abstractions that enable them to express various measurable properties of service antipatterns. Using a DSL offers greater flexibility than implementing *ad-hoc* detection algorithms manually because the DSL allows describing antipatterns using higher-level domain-related abstractions and focusing on *what* to detect instead of *how* to detect it [51]. Our DSL is independent of any implementation concern, such as the computation of static and dynamic metrics and, thanks to the unified meta-model, of the particular service technologies. Moreover, our DSL allows the adaptation of the antipattern specifications to the context and characteristics of the analysed SBS by adjusting the threshold used in the rules. For example, if the service interfaces of some SBS have, in general, high numbers of operations, engineers may choose to use the value of the `NOD` metric as `VERY_HIGH` instead of `HIGH`, as in Figure 10 for the *Multi Service* antipattern.

As for alternatives to our DSL, other rule-based declarative languages exist including the Object Constraint Language[2] (OCL). The OCL helps describing rules to apply on Unified Modeling Language (UML) models. However,

2. http://www.omg.org/spec/OCL

languages like the OCL do not suit our purpose because our specifications of service antipatterns comes with a higher level of abstraction and domain expressiveness. In addition, there exist not much pragmatic implementations for leveraging the OCL except the SimpleOCL[3], a not-fully-developed, proof-of-concept implementation of the OCL standard built on top of EMF and EMFText[4].

The unified meta-model in Figure 2 provides the vocabulary used in the BNF grammar of our DSL and its constructs, *e.g.*, *expressions* (service domain-specific) and *symbols* (approach-specific). The constructs fit our problem domain precisely. We combine those domain and approach-specific constructs to create a DSL that can be used to specify service antipatterns.

4.2 Generation of Detection Algorithms

This step follows a process to generate detection algorithms automatically, leveraging the model-driven engineering (MDE) methodology and using a template-based technique. The generative programming [52] is an approach for generating customised software artifacts including source code. Several works in the literature have leveraged generative programming, in particular, the template-based techniques, for design patterns or design antipatterns [11], [53], [54]. We define a unique template for all rule cards consisting of well-defined tags to be replaced with the values of different metrics defined in the rule cards.

From the previous step, using the specified rule cards, we generate detection algorithms for the service antipatterns by parsing the rule card and feeding templates with values extracted from the rule cards. For the generation of detection algorithms, we rely on the Eclipse Modeling Framework (EMF) [55] and its code-generation facility based on a predefined Ecore model [56]. The EMF project, a modeling framework, and code-generation facility, provides tools and runtime support to generate compilable code for the detection of each service antipattern.

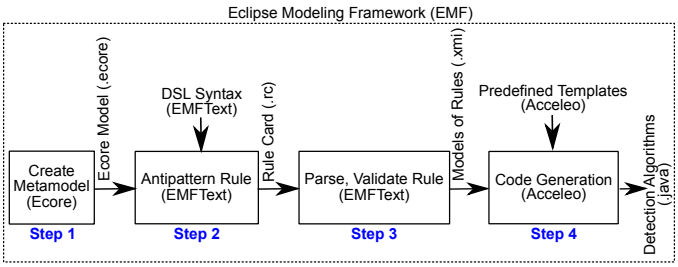


Fig. 6: Detection Algorithm Generation Process.

Figure 6 shows the different steps involved in the algorithm generation process. For each step in Figure 6, in the parentheses we show the technology/tools on which we rely, and, for each input/output, we show the file types within the parentheses. Following the generation process, first, we create a meta-model of our DSL as an Ecore instance (Step 1). We also define modules in the form of MTL files, which consists template(s). Our EMF meta-model consists of two parts, *e.g.*, the Ecore models and the genmodel description files. The Ecore file contains the information about the defined classes, *i.e.*, one executable main class for each service antipattern. The genmodel file contains additional information for the code generation, *e.g.*, the path and file information. In addition, the genmodel file could also have the control parameters, *i.e.*, how the code should be generated. In our approach, we have created static instances that are produced when we generate the code of our meta-model (*i.e.*, using a genmodel). We select the meta-model from which our generation module will take its types, in our case it is Ecore. We also need to choose the meta-class that will be used to generate the code file (in our case, EClass).

Then, we use EMFText [57] to write and validate rule cards on-the-fly (Step 2). For the generation of the detection algorithms, first, we parse the rule cards of each antipattern and represent them as models. Then, we use Ecore to validate syntactically the rule-card models against the meta-model of our DSL (Step 3). Ecore guarantees the correctness of the rule-card models. We use a template-based code generation technique provided by Acceleo [58] (Step 4). Acceleo is a pragmatic implementation of the Object Management GroupâĂŹs MOF Model to Text Language standard. An M2T (Model to Text) projectâĂŤAcceleoâĂŤfocuses on the generation of textual artifacts, *e.g.*, source code, from models. As the inputs, Acceleo needs a model in standard XML or XMI format and some templates (MTL) that we defined in Step 1. With these templates and the Acceleo engine, we parse the models of service antipatterns and generate the detection code.

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

4.3 SOFA Framework

We now describe UniDoSA underlying framework and its different components. Figure 7 shows the underlying framework, SOFA (Service Oriented Framework for Antipatterns), that supports the specification and detection of service antipatterns. SOFA is itself a SBS based on the SCA technology. It has eight components each of which provides a standalone service. The components include:

- ***Detection*** represents the main detection service that initiates and controls the overall detection process. It also provides an interface to the clients to run the detection process and helps the clients visualise the detection results;
- ***Metric*** provides the computation of both the static and dynamic metrics. This component also stores the static metric values in a repository to be used on the fly. The values of dynamic metrics cannot be stored, as they may change for different executions;
- ***Rule Specification*** is responsible for specifying and representing a rule card using the ***Rule*** and ***Operator***.

3. https://modeling-languages.com/simpleocl-tool/
4. http://www.emftext.org/index.php/EMFText

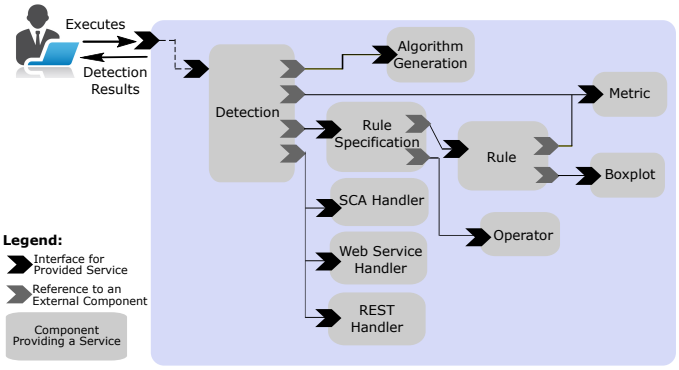


Fig. 7: SOFA Framework

All the rule cards are also stored in a repository used by the ***Algorithm Generation***;

- ***Algorithm Generation*** generates the detection algorithms automatically from the specified rule cards. These detection algorithms can be executed by the clients using the ***Detection*** service;
- ***Rule*** represents a repository of all the rules, a combination of which is known as a *rule card*. Some rules may depend on ***Metric*** to compute required metrics;
- ***Operator*** provides all the boolean and comparison operators to merge or group rules to form a rule card;
- ***Boxplot*** provides the means for computing boundary and threshold values. It provides all statistical analyses during the detection phase of our approach;

SOFA also has three other components: ***SCA Handler***, ***Web Service Handler***, and ***REST Handler*** dedicated to the analyses of different technology-specific systems. For example, the *REST Handler* component assists in sending HTTP requests and receiving HTTP responses to be analysed later on while the *Web Service Handler* assists for the SOAP technology to invoke concretely SOAP Web services to measure dynamic metrics.

The ***Detection*** component in the SOFA framework returns the occurrences of service antipatterns into a text file with the detection details, *e.g.*, the values of underlying metrics, the structural properties of service/component identified as being parts of some antipatterns, the box-plot values as compared to various other services/components in the analysed system. Engineers can use this output file to further manually investigate the occurrences, *i.e.*, to decide on how to refactor or resolve the antipatterns based on their knowledge and experience.

In the following, we discuss the detection of the *Multi Service* antipattern as our example.

5 A Detailed Example

In this section, we show an application of our UniDoSA approach for the specification and detection of service antipatterns in SBSs through the widely known *Multi Service* antipattern.

As shown in Figure 5, our approach consists of four steps. In the following, we show the application of our approach on the *Multi Service* antipattern.

5.1 Domain Analysis (Step 1)

The first step of UniDoSA involves domain analysis and specification for each service antipattern. The domain analysis is inspired from previous works [50]. It is "*a process by which information used in developing software systems is identified, captured, and organised with the purpose of making it reusable when creating new systems*" [50]. In our approach, service antipatterns are the *information* and the detection algorithms are *software systems*. The information on specified service antipatterns are to be reusable when specifying new service antipatterns. Thus, the domain analysis ensures that the language for specifying antipatterns is built upon consistent high-level abstractions and is flexible with high expressiveness. This step is crucial to UniDoSA because its output serves as the input for the following steps. As we identify the key concepts through the domain analysis, we can express them as properties-value pairs. This can be done using the following two steps:

5.1.1 Process

Input: Textual descriptions of service antipatterns from the literature, such as [9], [8], [19], [40], [39], [42], [49], [41]. We also require the meta-model and the general architectural description of each service technology.
Output: A list of the key attributes found in the literature to describe service antipatterns that forms a vocabulary for antipatterns. We also obtain concepts that works as the basis for building our unified meta-model. These base information consist of individual elements and their relationships within and across the service technologies.
Description: The goal of processing of the descriptions is to identify, define, and organise key attributes used to describe the antipatterns. The key attributes refer to keywords or specific service-oriented concepts used to describe antipatterns in the literature [8], [9], [19], [39], [40], [42], [49], [41]. These collected attributes carry a vocabulary of reusable concepts to specify antipatterns. For the domain analysis, we perform a thorough search of the literature for key concepts in the antipatterns descriptions. The analysis is performed iteratively. More specifically, for each description of an antipattern, we extract all key concepts, compare them with the collected concepts and, finally, include new concepts in the domain, taking into account synonyms and homonyms [59]. In the end, we realise a collection of concepts that form a unified vocabulary for specifying service antipatterns. The attributes can be of three types: measurable, lexical, or structural. Measurable attributes are countable (*e.g.*, interfaces, operations, and so on). The lexical attributes are the vocabulary used to name service artifacts like interfaces, operations, or service parameters. Structural attributes and relationships define the structures of service artifacts. We also combine three technology-specific meta-models and come up with a unified meta-model, the individual element of which are domain-specific concepts. We count or measure on the elements of this unified meta-model as shown in Table 3.
Implementation: This is essentially a manual step as indicated in Figure 5. However, this step requires the engineers' knowledge and expertise and can be assisted using domain analysis tools.

5.1.2 Example with Multi Service

We summarise the textual description of the *Multi Service* [8] antipattern in Table 2. In the description of the *Multi Service*, we identify the key concepts (in italic in the table) of services with many defined operations related to different business and technical abstractions, not easily reusable because of the low cohesion of their methods, often unavailable to end-users because they are overloaded and may have a high response time [8]. The measurable attributes of *Multi Service* include the concepts of *many* operations, *low* cohesion, *high* response time, and *low* availability.

We characterise the identified measurable attributes by values specified using keywords such as *high*, *low*, *few*, and *many*, as found in the textual description of the *Multi Service* antipattern. The attributes can be aggregated using common set operators, *e.g.*, intersection ($\cap$) and union ($\cup$). In our example, all attributes must be measured to identify a service as a *Multi Service* antipattern. We present more details on the attributes and their possible values for the key concepts in Section 4.1 where we present the DSL derived from our domain analysis, its grammar, and a list of the attributes and their possible values.

5.1.3 Discussion

The process of domain analysis is iterative. In particular, if we want to consider a new service antipattern, we must extract the key concepts or attributes from its description and compare them with existing attributes for avoiding repetition. In our domain analysis, we study 12 service antipatterns. However, other existing service antipatterns from the literature can be analysed and the DSL can be extended with new attributes and values. We believe these 12 service antipatterns are representative of the whole set of service antipatterns described in the literature and include more than 40 reusable key concepts. Using these key concepts, our DSL forms a consistent vocabulary of reusable concepts to specify service antipatterns. One of the outcomes in this Domain Analysis step is that we provide an identifier for each key concepts (which are measurable properties of elements in our meta-model) related to the 12 services antipatterns. These key concepts are also known as *metrics* in our DSL.

5.2 Specification (Step 1)

After we conduct the domain analysis, the specification of services of antipatterns follows. In this phase, we specify each service antipattern with a high-level of abstraction in a repetitive process.

5.2.1 Process

Input: A vocabulary of service antipatterns, *i.e.*, in our case the meta-model elements, their measurable attributes/properties, and their measurement values/scales.
Output: Specifications detailing the rules to apply on a SBS to detect the specified service antipatterns.
Description: We formalise the attributes required to specify detection rules at a high-level of abstraction using a DSL (see Figure 8). The DSL allows the specification of service antipatterns declaratively via the compositions of rules in rule cards (*i.e.*, sets of related rules), for example as shown in Figure 10 for the *Multi Service* antipattern. Using the antipattern vocabulary, we construct rules and rules cards for each service antipattern. In general, we put each measurable attribute in a rule, *e.g.*, the rule checks for the value of an attribute or if an attribute reaches/exceeds a predefined threshold value.

Individual rules may relate to the structure of the service interfaces or the behavior of services. For uniformity, we consider that antipatterns characterise services. Thus, a rule for detecting multi methods/operations reports the service interfaces defining a multitude of methods/operations.

A rule verifies and, for example, returns the set of services if the total number of methods/operations is too high or low compared to other services in the system. Our defined rules have a consistent granularity and thus it is possible to combine their outputs using common set operators, like union and intersection. In our UniDoSA approach, we choose services as the level of granularity for the sake of simplicity and without loss of generality.
Implementation: Based on their knowledge and experience, engineers manually define the specifications for the detection of service antipatterns using the vocabulary.

5.2.2 Example with Multi Service

We define our DSL using a *Backus Normal Form* (BNF) grammar as shown in Figure 8. A *rule card* is identified by the keyword `RULE_CARD`, followed by a rule card name and a set of rules (line 1). A rule describes a set of static or dynamic properties, *e.g.*, metrics (lines 9–12), and may have relationships with other rules, such as via association (`ASSOC`) (lines 17-19), and may combine with other rules via set operators such as union (`UNION`) or intersection (`INTER`) (line 6). A metric may define a numerical value (line 7) or an ordinal value defined using five-point Likert scale (line 13): very high, high, medium, low, very low. However, the values can be also boolean type (`TRUE` or `FALSE`, line 14) and reference type (`NULL` or `EMPTY`, line 15). In the SOFA framework, we define ordinal values by relating ordinal values with concrete numerical values to avoid manually setting threshold values using the box-plot statistical technique [60] as shown in Figure 9.

We identify and define a set of 41 metrics (see lines 9-12 in Figure 8) after the domain analysis, as listed in Tables 4 and 5. This extendable metric suite can be used to specify various service antipatterns in SCA, SOAP, and REST.

Figure 10 shows the rule cards for *Multi Service* and *Tiny Service* antipatterns. The *Multi Service* antipattern is characterised by very high response time, high number of operations, low availability of `SOAPService` or `SCAService`, and low cohesion among its `Method/Operation`s. A *Tiny Service* corresponds to a service that declares a very low number of operations and has a high coupling with other services. In Table 6, we define metrics relevant to *Tiny Service* and *Multi Service* antipatterns, namely `COH` - Cohesion (adapted from [61]), `CPL` - Coupling, `NOD` - Number of Operations Declared, `RT` - Response Time, and `A` - Availability.

Figure 11 shows the similarity between the *Multi Service* antipatterns in SCA and SOAP. As shown in Figure 10, the *Multi Service* antipattern characterises a service (or component) with high number of operations (or methods) defined in its interface, which corresponds to the same class, *i.e.*,

```
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 | id_metric comparator const_value

9  id_metric    ::= ALS | ANAM/ANAO | ANIM | ANP | ANPT | ARIM | ARIO | ARIP | COH | CPL | NCO | NI | NIR | NMD/NOD | NOPT
10                | NOR | NPT | NSE | NUM | NVMS | NVOS | RGTS | TNP
11                | AC | AK | CC | CCV | ET | HL | HM | LF | RRF | SC | SCV | TLB | VRB | VRU
12                | A | NMI | NTMI | RT

13 ordi_value   ::= VERY HIGH | HIGH | MEDIUM | LOW | VERY LOW
14 bool_value   ::= TRUE | FALSE
15 ref_value    ::= NULL | EMPTY

16 comparator   ::= < | ≤ | = | ≠ | ≥ | > | ∈ | ∉

17 relationship ::= relationType FROM ruleName cardinality TO ruleName cardinality
18 relationType ::= ASSOC | COMPOS
19 cardinality  ::= ONE | MANY | ONE_OR_MANY | num_value NUMBER_OR_MANY

20   rule_cardName, ruleName, ruleClass ∈ string
21   num_value ∈ double
22   const_value ∈ string
```

Fig. 8: BNF Grammar of Rule Cards in UniDoSA.

TABLE 4: List of Service Metrics for Specifying Service Antipatterns in SCA and SOAP. (Relevant concepts from the unified meta-model in Figure 2 are marked in **bold**.)

Metrics	Full Names	Static/Dynamic
A	Availability of a **Service**	dynamic
ALS	Average Length of **Signatures/Names**	static
ANP	Average Number of **Parameters** in **Operations**	static
ANPT	Average Number of Primitive Type **Parameters**	static
ANIO	Average Number of Identical **Operations**	static
ANAO	Average Number of Accessor **Operations**	static
ARIP	Average Ratio of Identical **PortTypes**	static
ARIO	Average Ratio of Identical **Operations**	static
ARIM	Average Ratio of Identical **Messages**	static
COH	**Service** Cohesion	static
CPL	**Service** Coupling	static
NCO	Number of CRUD **Operations**	static
NOD	Number of **Operations** Declared	static
NOPT	Number of **Operations** in **PortTypes**	static
NI	Number of **Interfaces**	static
NIR	Number of Incoming **References**	static
NMI	Number of **Method** Invocations	dynamic
NOR	Number of Outgoing **References**	static
NPT	Number of **PortTypes**	static
NTMI	Number of Transitive **Methods** Invoked	dynamic
NSE	Number of **Services** Encapsulated	static
NUM	Number of Utility **Methods**	static
NVMS	Number of Verbs in **Message Signatures/Names**	static
NVOS	Number of Verbs in **Operation Signatures/Names**	static
RGTS	Ratio of General Terms in **Signatures/Names**	static
RT	Response Time of a **Service**	dynamic
TNP	Total Number of **Parameters**	static

`Operation/Method`, in the unified meta-model in Figure 2. Moreover, the cohesion metric is calculated for both technologies using the `Operation/Method` and `Message` elements (*e.g.*, `InputMessage` and `OutputMessage`). In contrast, the two other characteristics, *e.g.*, availability and response time, correspond to `Component` and `SOAPService` for SCA and SOAP services, respectively. This difference comes from the different granularity and architectural building blocks of the two technologies.

Like for SCA and SOAP, the specifications of REST antipatterns also use metrics, which are measured from the *resources* (*i.e.*, `Resource` in the meta-model), perspective whereas the metrics in SCA and SOAP are mostly

TABLE 5: List of Service Metrics Specific for Antipatterns in REST. (Relevant concepts from the unified meta-model in Figure 2 are marked in **bold**.)

Metrics	Full Names	Static/Dynamic
AC	**Authentication** Cookie	dynamic
AK	Action Keywords	static
CC	Client Cookie	dynamic
CCV	Client **Caching** Value	dynamic
ET	Entity Tag	dynamic
HL	Header **Link**	dynamic
HM	**Http Method**	static
LF	Location Field	dynamic
RRF	**Resource Representation** Format	dynamic
SC	Server **Cookie**	dynamic
SCV	Server Caching Value	dynamic
TLB	Total Links in response **Body**	dynamic
VRB	Verbs in **Request Body**	dynamic
VRU	Verbs in **Request URI**	static

service interface and operation centric (`Specification` and `Method/Operation` in the proposed meta-model). We show 14 REST-specific metrics in Table 5. For example, as presented in Figure 12, *Forgetting Hypermedia* [44] represents a service in which the link to a `Resource` entity, *i.e.*, *entity links*, are absent in the response body or header provided by the response. For HTTP GET requests, such *entity links* should be provided in the response body or header, hence, checking missing links in the body or header is adequate (see line 9-10 in Figure 12). For HTTP POST requests, the server should provide an external *location* in the response header or a *link* in the response body. Therefore, looking for the absence of such location in the response header (see line 11 in Figure 12) or missing link in the response body (see line 9 in Figure 12) is enough to detect *Forgetting Hypermedia* service antipattern in REST.

5.2.3 Discussion

Carrying out the domain analysis ensures that the specifications of service antipatterns are done with high-level abstractions and captures domain expertise and reusable

knowledge. Moreover, our DSL provides greater flexibility than implementing ad-hoc detection algorithms. Domain experts can modify the specifications at a high-level of abstraction without complete knowledge of the underlying detection framework, for example, either by adding new rules or by modifying existing ones, or by modifying the threshold values for metrics. Our proposed DSL is precise (*i.e.*, no ambiguity) and expressive. In fact, the use of our DSL is not dependent on computer skills or knowledge about our underlying SOFA framework. The rule cards for all the 12 services antipatterns are available in our technical report [62].

The domain analysis and specification are iterative: if during the process a key concept or an attribute is left unconsidered, it can be added to our DSL later. Our method as well as our specification language are flexible and the flexibility relies on the expressiveness of the language and available key concepts that we verified on a set of 12 service antipatterns.

5.3 Generation of Detection Algorithm (Step 2)

This section presents the generation of the detection algorithm for the *Multi Service* antipattern for the sake of completeness. We put more technical details in our technical report [62].

5.3.1 Process

Input: Rule cards of service antipatterns.

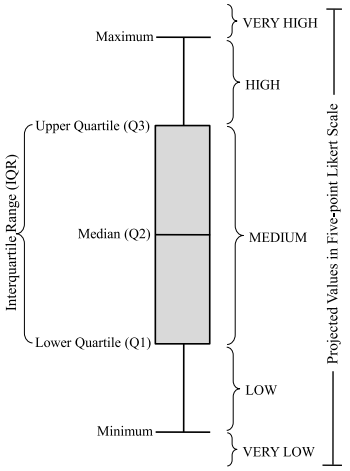


Fig. 9: Numerical Values Projected to Five-point Likert Scale Ordinal Values using a Boxplot.

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

```
1 RULE_CARD: TinyService {
2  RULE: TinyService {INTER FewOperation HighCoupling};
3  RULE: FewOperation {NOD VERY_LOW};
4  RULE: HighCoupling {CPL HIGH};
5 };
```

Fig. 10: Rule Cards for *Multi Service* and *Tiny Service* Antipatterns in SCA and SOAP services.

TABLE 6: The Definitions of Metrics in *Multi Service* and *Tiny Service* Antipatterns.

Metric: Cohesion (COH) **COH = (SIDC + SIUC + SISC) / 3**, where, SIDC (Service Interface Data Cohesion) = Common(Param($SO_p(si_s)$)) + Common(returnTypes($SO_p(si_s)$)) / Total($SO_p(si_s)$) $\times$ 2, where, - $SO_p(si_s)$, set of all operations exposed in the interface si_s of service s; - Common(Param($SO_p(si_s)$)), a function to calculate the number of service operation pairs with at least one common input parameter type; - Common(returnTypes($SO_p(si_s)$)), a function to calculate the number of service operation pairs with the same return type; - Total($SO_p(si_s)$), a function to return the number of all possible combinations of operation pairs for the service interface si_s; SIUC (Service Interface Usage Cohesion) = Invoked(clients, $SO_p(si_s)$) / (|clients| $\times$ |$SO_p(si_s)$|), where, - clients, the set of all the clients of service s; - Invoked(clients, $SO_p(si_s)$), a function to compute the total number of all used operations by a client; SISC (Service Interface Sequential Cohesion) = SeqConnected($SO_p(si_s)$) / Total($SO_p(si_s)$), where, - SeqConnected($SO_p(si_s)$), a function to calculate the number of service operation pairs that have sequential dependencies; - Total($SO_p(si_s)$), a function returning the number of all possible combinations of operation pairs; **Scale:** Absolute; **Measurement Unit:** Count; **Range:** between 0 and 1;
Metric: Coupling (CPL) **CPL = CPL$_{indv}$ / CoupleSum(CPL$_{indv}$)**, where, - CPL$_{indv}$ = NIC + NOC for each service or component; - NIC, total number of incoming connections for a service or component; - NOC, total number of outgoing connections for a service or component; - CoupleSum(CPL$_{indv}$), a function to calculate the total incoming and outgoing connections for all individual services or components; **Scale:** Absolute; **Measurement Unit:** Count; **Range:** between 0 and 1;
Metric: Number of Operations Defined (NOD) **NOD = Count(Op$_{si}$)**, where, - Count(Op$_{si}$), a function returning the total number of operations or methods defined in a service interface (si); **Scale:** Absolute; **Measurement Unit:** Count; **Range:** minimum 1;
Metric: Response Time (RT) **RT = Sum(Time$_{res_rcvd}$ - Time$_{req_sent}$) / TotalOpInvoked$_{si}$**, where, - Sum(Time$_{res_rcvd}$ - Time$_{req_sent}$), a function to calculate the total time required to receive responses for all the invoked operations from a service interface; - TotalOpInvoked$_{si}$, total number of operations invoked from a service interface; **Scale:** Absolute; **Measurement Unit:** Milliseconds; **Range:** minimum 0;
Metric: Availability (A) **A = TotalSuccessResponse$_{si}$ / TotalOpInvoked$_{si}$**, where, - TotalSuccessResponse$_{si}$, total number of success responses received after sending an arbitrary number of service requests; - TotalOpInvoked$_{si}$, total number of operations invoked from a service interface; **Scale:** Ratio; **Measurement Unit:** Percentage; **Range:** between 0 and 100;

Output: Automatically generated detection algorithms for the service antipatterns.

Description: From our textual unified DSL, we build a meta-model for DSL using the Ecore [56] (as shown in our technical report [62]) and a parser to model rule cards and manipulate these models of service antipatterns programmatically. Then, we automatically generate algorithms using our defined templates. The detection algorithms are based both on the models of the service antipatterns and

the underlying SOFA framework. The generated detection algorithms are correct by construction of our specifications using a DSL.

Implementation: The generation of detection algorithms is automatic and relies on our SOFA framework (Service Oriented Framework for Antipatterns) that provides services required by all detection algorithms. As discussed in Section 4.3 and shown in Figure 7, these services implement operations related to the operators (*i.e.*, how the set operations are performed on the results of two or multiple rules through the `Operator` service), attributes (*i.e.*, how the metric values are calculated using the `Metric` service), and the ordinal values (*i.e.*, how the `Boxplot` service computes and maps the numerical values to the ordinal values). Our SOFA framework also provides services to handle technology-specific building blocks and components. For example, the parsing of the WSDL specification files for SOAP services is different to parsing SCDL specification files in SCA systems. In summary, our SOFA framework can automatically compute metrics, perform structural analyses on service interfaces, and apply the rules on these services to decide if a service can be considered as an antipattern. Our set of ten services (see Figure 7) and the overall design of the SOFA framework are driven by the key attributes from the domain analysis and support our proposed DSL. For the sake of brevity, we present and describe the meta-model of rule cards that is achievable from our DSL in our technical report [62].

Framework for Detection: Our SOFA framework is developed upon the component-based SCA technology. We chose the SCA as our implementation choice because of its advantages over other technologies including (1) flexibility of development and (2) reusability, *i.e.*, each service component has well-defined interfaces, thus, it can be developed, tested and debugged independently.

We rely on OW2 FraSCAti [63] as our framework underlying technology. OW2 FraSCAti was itself implemented as an SCA application composed of 13 SCA composites

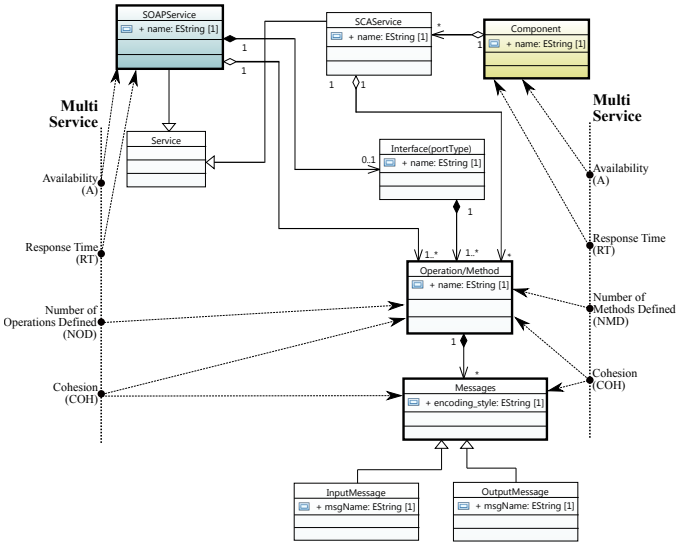


Fig. 11: Similarity between *Multi Service* Antipatterns in SCA (right) and SOAP (left).

```
1  RULE_CARD: ForgetHyperMedia {
2    RULE: ForgetHyperMedia {UNION GetRequestLink PostRequestLink};
3    RULE: GetRequestLink {INTER HttpMethodGet NoLinkGet};
4    RULE: PostRequestLink {INTER HttpMethodPost NoLinkPost};
5    RULE: NoLinkGet {UNION NoBodyLink NoHeaderLink};
6    RULE: NoLinkPost {INTER NoBodyLink NoLocationHeader};
7    RULE: HttpMethodGet {HM = 'GET'};
8    RULE: HttpMethodPost {HM = 'POST'};
9    RULE: NoBodyLink {TLB = 0};
10   RULE: NoHeaderLink {HL = NULL};
11   RULE: NoLocationHeader {'Location' ∉ ResponseHeader};
12 };
```

Fig. 12: Rule Card of the *Forgetting Hypermedia* REST Antipattern.

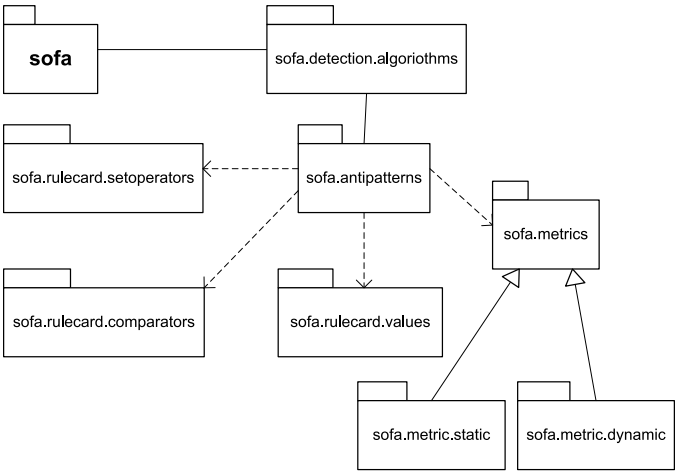


Fig. 13: Implementation Architecture of the SOFA Framework.

(with a total of 91 SCA components). The OW2 FraSCAti can encompass service components and bindings from various service technologies including JAX-WS, JMS, EJB, RMI, and REST. Thus, the benefit of using FraSCAti lies in that applications built with diverse technologies and bindings could be introspected and reconfigured at runtime via its own APIs. For example, the OW2 FraSCAti Explorer APIs enable loading and starting an SCA specification composite file, exploring an already running SCA application then discovering underlying SCA components and discovering SCA services and references to other components. It can also start or stop the components or composites dynamically. However, for the services under analysis from any service technology, we must build a service skeleton imitating the real service interfaces. Figure 13 shows the implementation architecture of our SOFA framework. In the following, we show some code snippets related to the implementation part of our SOFA framework.

Set Operators: Our operator packages are defined under the `ofa.rulecard.setoperators` package that defines the methods required to perform intersection or union between the results returned by the individual rules, *i.e.*, a set of services satisfying or reaching defined metric thresholds. We apply these operators on the sets of services that are potential antipattern candidates. The resultant set contain only the appropriate services. For example, the `exec()` method in Listing 1 performs an intersection on two sets of services and returns a new set of services. To ease the

set operations, within a set we only provide signatures of services.

Listing 1: Code Snippet of `Intersection` Operator.

```
public class Intersection implements ASetOperator {

 @Override
 public Set<String> exec(Set<String> set1,
   Set<String> set2) {
   return Sets.intersection(set1, set2);
 }
}
```

Measurable Properties: We compute metrics as measurable properties using the introspection features provided by the OW2 FraSCAti APIs. Our metric suit consists of 41 measurable properties as discussed in Section 5.2. Our SOFA framework can compute any metric on a set of services. For example, in the Listing 2 below, the `compute()` method computes the metric `NMD` (also known as `NOD`) on each service of a system.

Listing 2: The Class Calculating the `NMD` Metric on a Service Interface.

```
public class NMD extends AMetric {

@Override
public void compute(Object relService, String serviceName,
Component currentComponent, String componentName) {
   if (!this.boxplot.getEntries().containsKey(serviceName)){
   this.boxplot.addEntry(componentName,
   new relService.getClass().getDeclaredMethods().length);
   }
}
}
```

The ***Boxplot*** service component of our SOFA framework offers methods to compute and access the quartiles for and outliers of a set of metric values as illustrated in Listing 3.

Listing 3: The Class Defining the Boxplot Service of our SOFA Framework.

```
public class BoxPlot {
 public double getHigherOutlier(){
   return this.sortedValues[this.nbValues-1];
 }
 public double getLowerQuartile() {
   int lowerQuartile = (this.nbValues + 1) / 4;
   return this.sortedValues[lowerQuartile];
 }
 public double getUpperQuartile() {
   int upperQuartile = (3 * this.nbValues + 3) / 4;
   return (upperQuartile < this.sortedValues.length) ?
        this.sortedValues[upperQuartile] : 0.0;
 }
 public Map<String, Double> getHighOutliers() {
   return this.getValues(">", this.maxBound -
        this.fuzziness, null);
 }
 public Map<String, Double> getLowOutliers() {
   return this.getValues("<=", this.minBound +
        this.fuzziness, null);
 }
}
```

Algorithm Generation: We start with the rule card of a service antipattern modelled using Ecore. The Ecore model can also be represented in XMI as shown in the below:

Listing 4: Rule Card Model for *Multi Service* Antipattern.

```
<?xml version="1.0" encoding="ASCII"?>
<rulecards:RuleCard xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:rulecards="http://rulecards/1.0" name="MultiService">
 <children xsi:type="rulecards:SmellComposite"
      name="MultiService" children="//@children.1
      //@children.2 //@children.3 //@children.4"/>
 <children xsi:type="rulecards:SmellOrdinalValue"
      name="MultiOperation" comparator="EQUAL"
      ordinalValue="VERY_HIGH">
   <leftMetricValue xsi:type="rulecards:MetricLeaf"
        metric="NOD"/>
 </children>
 <children xsi:type="rulecards:SmellOrdinalValue"
      name="HighResponse" comparator="EQUAL"
      ordinalValue="VERY_HIGH">
   <leftMetricValue xsi:type="rulecards:MetricLeaf"
        metric="RT"/>
 </children>
 <children xsi:type="rulecards:SmellOrdinalValue"
      name="LowA" comparator="EQUAL" ordinalValue="LOW">
   <leftMetricValue xsi:type="rulecards:MetricLeaf"
        metric="A"/>
 </children>
 <children xsi:type="rulecards:SmellOrdinalValue"
      name="LowCohesion" comparator="EQUAL"
      ordinalValue="LOW">
   <leftMetricValue xsi:type="rulecards:MetricLeaf"
        metric="COH"/>
 </children>
</rulecards:RuleCard>
```

We implement the generation of the detection algorithms using an approach similar to that proposed in our previous work [11], which relies on the Visitor pattern [64]. We benefit from the Visitor because, for each antipattern, using the same Java template, we can generate varied execution code that are detection algorithms. As discussed earlier, our templates are predefined snippets of Java code with well-defined tags that can be replaced by concrete values at runtime. More details on the templates and generation algorithm can be found in our technical report [62].

5.3.2 *Example with Multi Service*

The following code snippet presents the visit method to generate the detection rule related to a measurable property or metric. When we visit the model of the rule card (as shown in Listing 4, we replace the tag `<CODESMELL>` by the name of the rule. For example, for *Multi Service*, we replace the tag `<CODESMELL>` with *MultiOperation* (see Figure 10). We also replace the tag `<METRIC>` by the name of the metric. For example in Figure 10, we replace the tag `<METRIC>` by `NOD`. Finally, we replace the tag `<ORDINAL VALUES>` by an ordinal value. For example for the metric `NOD`, it is `VERY_HIGH`, as shown below in Listing 5.

Listing 5: The Visitor Method to Generate the Detection Rule of a Metric.

```
public void visit(IMetric aMetric) {
 replaceTAG("<CODESMELL>", aRule.getName());
 replaceTAG("<METRIC>", aMetric.getName());
 replaceTAG("<ORDINAL_VALUE>", aMetric.getOrdinalValue());
}

private String getOrdinalValue(int value) {
 String method = null;
 switch (value) {
   case VERY_HIGH : method = "getHighOutliers";
   break;
   case HIGH : method = "getHighValues";
   break;
   case MEDIUM : method = "getNormalValues";
   break;
   case VERY_LOW : method = "getLowOutliers";
   break;
   case LOW : method = "getLowValues";
   break;
   default : method = "getNormalValues";
   break;
```

```
22  }
23 return method;
24}
```

The detection algorithm for a service antipattern is defined as implementing the interface, `ServiceAntipattern`, as shown in Listing 6.

Listing 6: The Automatically Generated Detection Algorithm for *Multi Service* Antipattern.

```
1public class MultiService extends ServiceAntipattern {
2
3  public MultiService() {
4
5    MetricValue metricValue1Smell1 = new
         MetricLeaf(Metric.NMD);
6    Smell smell1 = new SmellOrdinalValue(metricValue1Smell1,
         Comparator.EQUAL, OrdinalValue.VERY_HIGH);
7
8    MetricValue metricValue1Smell2 = new
         MetricLeaf(Metric.COH);
9    Smell smell2 = new SmellOrdinalValue(metricValue1Smell2,
         Comparator.EQUAL, OrdinalValue.LOW);
10
11   MetricValue metricValue1Smell3 = new
         MetricLeaf(Metric.RT);
12   Smell smell3 = new SmellOrdinalValue(metricValue1Smell3,
         Comparator.EQUAL, OrdinalValue.VERY_HIGH);
13
14   MetricValue metricValue1Smell4 = new
         MetricLeaf(Metric.A);
15   Smell smell4 = new SmellOrdinalValue(metricValue1Smell4,
         Comparator.EQUAL, OrdinalValue.LOW);
16
17   this.rootSmell = new SmellComposite(SetOperator.INTER);
18   this.rootSmell.addChildSmell(smell1);
19   this.rootSmell.addChildSmell(smell2);
20   this.rootSmell.addChildSmell(smell3);
21   this.rootSmell.addChildSmell(smell4);
22  }
23}
```

The final detection algorithm aggregates the detection algorithms of several metrics, implementing the interface `MetricValue` as shown in Listing 7.

Listing 7: Combining the Computation of Individual Metric using Set Operators.

```
1public class MetricLeaf implements MetricValue {
2
3  public MetricLeaf(Metric metricId) {
4    this.metricId = metricId;
5  }
6
7  @Override
8  public BoxPlot getBoxPlot() {
9    return this.metricId.getImpl().getBoxplot();
10 }
11
12 @Override
13 public Set<Metric> getMetrics() {
14   Set<Metric> set = new HashSet<>();
15   set.add(metricId);
16   return set;
17 }
18
19 @Override
20 public String toString() {
21   return metricId.name();
22 }
23}
```

We combine the results of the computation of individual metric using the set operators of our ***Boxplot*** service to obtain occurrences of the antipattern.

5.3.3 Discussion

The Ecore models derived from our meta-model, DSL, and SOFA framework provide a concrete mechanism to generate and apply detection algorithms automatically. The addition of another attribute or metric in the DSL requires the implementation of the analysis within SOFA. In particular, the implementation of the computation of the metric. This addition of new metrics might vary from minutes to hours depending on the complexity of the metric being added. The Ecore models of service antipatterns that are used to generate detection algorithms must be instantiated each time for each antipattern. In our UniDoSA approach, all the metric values are computed on the fly.

5.4 Detection (Step 3)

This section briefly discusses how we perform the detection of an antipattern.

5.4.1 Process

Input: The detection algorithms for service antipatterns and a service-based system in which to detect the antipatterns.
Output: Suspicious services whose interfaces and behaviour conform to the specifications of service antipatterns.
Description: We automatically apply the detection algorithms on service-based systems to detect suspicious services. Detection algorithms can be applied individually to detect a single service antipattern or in batch to detect multiple service antipatterns at once.
Implementation: Invoking and applying the generated detection algorithms on a service-based system is straightforward, using the *Detection* service provided by our SOFA framework.

5.4.2 Example with Multi Service

Following our example of the *Multi Service* antipattern, we apply the detection algorithm of the *Multi Service* antipattern on, for example, the SCA specification along with the executable Jars of the service-based system for the SCA technology. Listing 8 shows a code snippet of the detection module with two SCA applications *HomeAutomation* and *FraSCAti*.

Listing 8: Main Detection Module for Service Antipatterns.

```
1public class Launcher {
2
3  public enum App {
4    HomeAutomation,
5    FraSCAti;
6    ...
7  }
8
9  public static App currentApp = App.FraSCAti;
10 public static SOFAManager sofaManager = new SOFAManager();
11 public static void main(String[] args) throws Exception {
12
13   for (Motif antipattern : Motif.antiPatterns())
14     sofaManager.addAntipatternToDetection(antipattern);
15
16   if (currentApp == App.HomeAutomation)
17     sofaManager.runDetection("jar", "composite");
18
19   else if (currentApp == App.FraSCAti)
20     sofaManager.runDetectionOfFraSCAti();
21
22   else if (currentApp == ...)
23     sofaManager.runDetectionOf...();
24
25   String metrics = sofaManager.getUsedMetricsResults();
26   System.out.println(metrics);
27   String detection = sofaManager.getDetectionResults();
28   System.out.println(detection);
29 }
30}
```

5.4.3 Discussion

Our UniDoSA approach can automatically analyse (1) specification files for SCA, *i.e.*, SCDL files with composites and components, (2) WSDL files with services, operations, and messages for SOAP services. For REST services, we perform a semi-automatic conversion of REST services into an SCA-compliant format with which we can concretely invoke the REST services using required credentials. SOFA framework should be extended to support the automatic analysis of the WSDL 2.0 standard, which is a *de-facto* specification file for REST services. We let such an extension as our future work.

In the following sections, we present experiments using SCA applications, SOAP Web services, and REST services.

6 Experiments

We now present a series of experimental studies through which we validate our UniDoSA approach. The goals of our experiments are to show that we can efficiently and effectively: (1) specify service antipatterns using the proposed DSL and that our DSL is extensible for new antipatterns and service technologies and (2) detect occurrences of the specified antipatterns across various service technologies efficiently in terms of the accuracy and performance of the detection algorithms.

6.1 Our Conjectures

Our experimental studies aim at supporting the following four conjectures:

- ***Generality*****:** *Our DSL allows the specification of different service antipatterns, from simple to more complex ones, independent of service technologies (SOAP, SCA, REST) using the proposed unified meta-model.*
- ***Accuracy*****:** *Our automatically-generated detection algorithms have a precision and recall greater than 75%, i.e., more than three-quarters of the detected occurrences of the antipatterns are true positive and more than three-quarters of all existing antipatterns are detected, respectively.*
- ***Extensibility*****:** *Our proposed unified meta-model, the service DSL, the proposed framework, SOFA, are extensible for adding new service metrics and technology-specific or technology-neutral antipatterns.*
- ***Performance*****:** *The average computation time required for the detection of service antipatterns using the generated algorithms is reasonably low, i.e., in the order of few seconds no matter the technology.*

6.2 Subjects

We use 12 service antipatterns from SCA, SOAP, and REST from the literature [65], [9], [39], [40], [8], [41], [42], which are commonly found and well-explained with related examples. Table 2 presents the list of service antipatterns that we analysed and detected.

6.3 Objects

We detect occurrences of the 12 service antipatterns in the following systems:

- **REST:** We use 18 widely-used and popular REST services that we found well-documented and listed in Table 7. These 18 REST define clearly their underlying HTTP methods, service end-points, authentication details, and client-request parameter details.
- **SCA:** We perform experiments on two SCA systems: *Home-Automation* [36] and *FraSCAti* [63]. The *Home-Automation* is developed independently in Spirals Project-Team, INRIA, France, to simulate a digital home for controlling basic household tasks remotely. *Home-Automation* is designed with 15 SCA components (including two extended components), each providing unique services with seven use cases. *FraSCAti* has a total of 91 SCA components providing 130 distinct services. To the best our knowledge, *FraSCAti* is currently the largest open-source SCA system implementing the SCA standard. We thus analyse in total 150 SCA services.
- **SOAP services:** Most of the SOAP Web services are proprietary. It is difficult to find freely-available services for our experimental studies. Web services search engines, *e.g.*, `eil.cs.txstate.edu/ServiceXplorer` or `www.programmableweb.com`, help finding service-interfaces and are limited in number and have the risk of providing broken or dislocated results. We perform our experiments with more than 120 SOAP services collected from the Programmable Web, *e.g.*, `www.programmableweb.com`, which is more updated and list more than 16,000 APIs of diverse technologies from wide range of service providers.

We believe that our experiments are reproducible on other services, in particular from the SCA, SOAP, and REST domains. We demonstrate the strength of our UniDoSA approach and the capability of the SOFA framework with two SCA systems, more than 120 of SOAP services, and 18 REST APIs. However, for the SCA systems, an executable package and its specification file in SCDL language are required for the detection. For SOAP services our approach requires a valid WSDL, and, for the REST, we require a list of resources and related HTTP verbs together with a base URI. With this minimal set of input, one can replicate our detection steps.

6.4 Process

Using the SOFA framework, we generate the detection algorithms corresponding to the rule cards of the 12 service antipatterns. Then, we apply these detection algorithms on the target SBSs. Finally, we validate the detection results by analysing the candidate service antipatterns manually (1) to validate that these candidate service antipatterns are true positives and (2) to identify false negatives (if any), *i.e.*, missing antipatterns.

To validate the results on REST and SOAP services, we hire two graduate students who independently validate the detected occurrences. We consolidate the results and come to conclusion for the occurrences where the two graduate students do not agree through discussions among all the authors and the students, finally using a maximum vote. As for the validation on *FraSCAti*, the core developers of the FraSCAti team assist us. We send our obtained detection

TABLE 7: List of 18 REST APIs and URLs of their Documentations.

REST APIs	Online Documentations
Alchemy	alchemyapi.com/api
BestBuy	developer.bestbuy.com
Bitly	dev.bitly.com/api.html
CharlieHarvey	charlieharvey.org.uk/about/api
DropBox	www.dropbox.com/developers/core/docs
Externalip	api.externalip.net
Facebook	developers.facebook.com/docs/graph-api
GoogleBook	developers.google.com/books
Instagram	instagram.com/developer
Linkedin	developer.linkedin.com/docs
Musicgraph	developer.musicgraph.com/api-docs/overview
Ohloh	github.com/blackducksw/ohloh_api
StackExchange	api.stackexchange.com/docs
TeamViewer	integrate.teamviewer.com/en/develop/api
Twitter	dev.twitter.com/overview/documentation
Walmart	developer.walmartlabs.com
YouTube	developers.google.com/youtube/v3
Zappos	developer.zappos.com/docs/api-documentation

results of a set of suspicious SCA components and the textual descriptions of service antipatterns to the core FraSCAti development team. Based on their pragmatic knowledge and development experiences, they manually evaluate each occurrence. Both for the SOAP services and the SCA detection results, we do not provide our specifications of service antipatterns not to bias them with our own understanding on service antipatterns.

We use precision, recall, and F_1-measure to measure the accuracy of our detection algorithms. Precision estimates the ratio of true occurrences identified among the detected antipatterns (*cf.* Equation 1) while recall estimates the ratio of detected antipatterns among the existing antipatterns (*cf.* Equation 2). F_1-measure is the harmonic mean of precision and recall to conclude on the detection accuracy with a single value (*cf.* Equation 3).

$$precision = \frac{|\{existing\_antipatterns\} \cap \{detected\_antipatterns\}|}{|\{detected\_antipatterns\}|} \quad (1)$$

$$recall = \frac{|\{existing\_antipatterns\} \cap \{detected\_antipatterns\}|}{|\{existing\_antipatterns\}|} \quad (2)$$

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \quad (3)$$

6.5 Detection Results and Discussions

Figures 14, 15, 16, and 17 present the detection results graphically using mosaic plots for the 12 service antipatterns in the 18 REST services, the FraSCAti and Home-Automation SCA systems with 150 services, and in 122 SOAP services. In total, the SCA-based *Home-Automation* system has 15 SCA components. We show the details on all the components in our technical report [62]. In Figure 16, we only show the SCA components in *Home-Automation* system detected as candidate service antipatterns by UniDoSA. Similarly, due to large number of SOAP services analysed in this paper, we only present in Figure 17 the SOAP services detected as candidate service antipatterns by UniDoSA. In the Figures

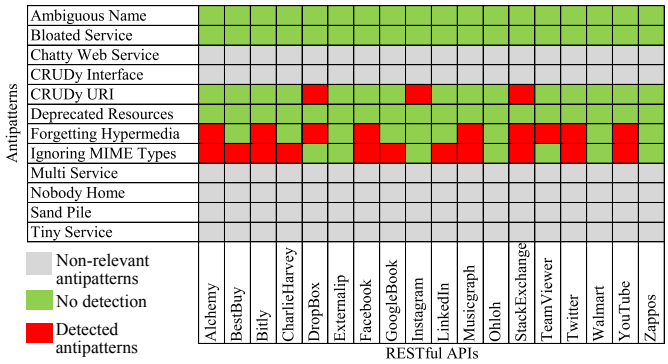


Fig. 14: Detection Results of 12 Antipatterns in 18 RESTful APIs.

14 to 17, the red squares present antipatterns detected for a service or component under analysis, the green squares represent no detection, and the grey squares indicate when a service antipattern is not applicable for a service technology.

Table 8 shows the detailed detection results where the first column lists various service antipatterns followed by the applicable technologies in the second column. The third and fourth column list candidate service (or component) antipatterns and related metric values or occurrences, respectively. We show the average detection time for each antipattern in the fifth column. The final three columns show the precision, recall, and F_1-measure for each antipattern. In the below, we discuss several detected service antipatterns in detail.

As Table 8 shows, *Ambiguous Name* is the most commonly occurred antipattern in the service domain. This indicates that designers and developers do not pay significant attention to properly naming the software entities like the services, interfaces, components, operations, and messages, and so on. This might hinder the reusability and understandability of, for example, service interfaces by the service consumers. Similarly, we found *Tiny Service* antipattern as more common than its counterpart *Multi Service* antipattern. This indicates that developers are aware of not overloading a service with a multitude of operations, rather, they tend to put very few operations under a service.

As for the REST-specific antipatterns, the results show that REST API designers often do not put hyperlinks in the response body or header, and thus, introduce the *Forgetting Hypermedia* antipattern. However, it is, in general, a good practice to provide links in the response, which promotes the HATEOS (Hypermedia as the Engine of Application State) constraint in REST. Furthermore, REST developers often tend to ignore representing resources in multiple formats to avoid development complexity. For example, REST service consumers may request a resource in JSON or XML or HTML or in any other formats, but the problem arises when the service developer strictly provides resources in one format (*e.g.*, XML), limiting the consumersâĂŹ choice. Due to this very common but poor practice, as shown in Table 8, we identified a high number of occurrences (94 out of 115 tested REST methods, *i.e.*, more than 80%) of *Ignoring MIME Types* antipatterns.

In the following sections, we present a detailed discuss

TABLE 8: General Detection Results of 12 Antipatterns.

Service Antipatterns	Applicable Technology	Identified Service(s)	Metrics/Occurrences	Average Detection Time	Precision	Recall	F_1
Ambiguous Name	SOAP	AIP3 PV ImpactCallback	ALS=0.675;RGTS=0.85;NVMS=26;NVOS=7;	0.855s	[9/9] 100%	[9/9] 100%	100%
		Bliquidity	ALS=0.576;RGTS=0.682;NVMS=42;NVOS=7;				
		CurrencyServerWebService	ALS=0.136;RGTS=0.682;NVMS=42;NVOS=5;				
		… …	… …				
		ProhibitedInvestorsService	ALS=0.158;RGTS=0.684;NVMS=12;NVOS=4;				
	SCA	*none detected*	*n/a*				
	REST	*none detected*	*n/a*				
Bloated Service	SOAP	*none detected*	*n/a*	0.071s	[3/3] 100%	[3/3] 100%	100%
	SCA *(FraSCAti)*	component-factory	NOI=1;NMD=7;TNP=12;COH=0.066;				
		factory	NOI=1;NMD=7;TNP=12;COH=0.066;				
		frascati-binding-http	NOI=1;NMD=5;TNP=8;COH=0.065;				
	REST	*none detected*	*n/a*				
Multi Service	SCA *(Home-Automation)*	IMediator	COH=0.027;NMD=13;RT=132ms;	78.67s	[2/2] 100%	[2/3] 66.67%	74.63%
	SCA *(FraSCAti)*	juliac	COH=0.1;NMD=5;RT=1018ms;				
		Explorer-GUI	*n/a*				
	SOAP	*none detected*	*n/a*				
Tiny Service	SCA *(Home-Automation)*	MediatorDelegate	NOR=4;CPL=0.44;NMD=1;	0.194s	[6/7] 85.71%	[6/6] 100%	92.31%
	SCA *(FraSCAti)*	sca-parser	NMD=1;CPL=0.56;	0.067s			
	SOAP	SrtmWsPortType	NOD=2;COH=0.0;	0.945s			
		Hydro1KWsPortType					
		ShadowWsPortType					
		XigniteTranscripts	NOD=4;COH=0.125;				
		BGCantorUSTreasuries	NOD=3;COH=0.083;				
Nobody Home	SCA *(Home-Automation)*	UselessService	NIR>0;NMI=0;	0.606s	[4/5] 80%	[4/4] 100%	88.89%
	SCA *(FraSCAti)*	NativeCompiler	NIR>0;NMI=0;				
		ServletManager	NMI=0;NIR>0;				
		WsdlCompiler	NMI=0;NIR>0;				
		BPELEngine	NMI=0;NIR>0;				
Deprecated Resources	REST	*none detected*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*
CRUDy Interface	SOAP	ForeignExchangeRates	COH=0.16;ANAO=66.67;NOD=24;RT=3113ms;NCO=9;	37.23s	[6/7] 85.71%	[6/6] 100%	92.31%
		TaarifCustoms	COH=0.12;ANAO=72.22;NOD=18;RT=4105ms;NCO=18;				
CRUDy URI	REST	DropBox (1 occurrence)	*e.g.,* POST /fileops/move				
		Instragram (1 occurrence)	*e.g.,* GET /users/search				
		StackExchange (3 occurrences)	*e.g.,* GET /me/write-permissions				
Sand Pile	SCA *(Home-Automation)*	HomeAutomation	NCS=13;ANP=1;ANPT=1;ANAM=100%;COH=0.17	0.184s	[1/1] 100%	[1/1] 100%	100%
Chatty Web Service	SOAP	ForeignExchangeRates	COH=0.16;ANAO=50;NOD=24;RT=3286ms;	1.89s	[1/2] 50%	[1/1] 100%	66.67%
		TaarifCustoms	COH=0.12;ANAO=72.22;NOD=18;RT=4105ms;				
Forgetting Hypermedia	REST	Alchemy (1 occurrence)	*e.g.,* GET /calls/url/URLGetRankedNamedEntities	19.54s	[53/55] 96.36%	[53/53] 100%	98.16%
		Bitly (2 occurrences)	*e.g.,* GET /shorten				
		Dropbox (10 occurrences)	*e.g.,* GET /metadata/{root}/{path}				
		Facebook (8 occurrences)	*e.g.,* GET /{objet-id}/comments				
		Musicgraph (7 occurrences)	*e.g.,* GET /api/v2/artist/search				
		StackExchange (17 occurrences)	*e.g.,* GET /me/questions/no-answers				
		TeamViewer (3 occurrences)	*e.g.,* GET /api/v1/groups/{ID}				
		Twitter (4 occurrences)	*e.g.,* GET /application/rate_limit_status.json				
		YouTube (3 occurrences)	*e.g.,* GET /playlists				
Ignoring MIME Types	REST	Alchemy (2 occurrences)	*e.g.,* GET /calls/url/URLGetRankedNamedEntities	19.39s	[94/94] 100%	[94/94] 100%	100%
		BestBuy (1 occurrence)	*e.g.,* GET /v1/products				
		Bitly (3 occurrences)	*e.g.,* GET /link/clicks				
		CharlieHarvey (4 occurrences)	*e.g.,* GET /page/api/recent				
		Facebook (2 occurrences)	*e.g.,* GET /{id}/friends				
		GoogleBook (1 occurrence)	*e.g.,* GET /volumes				
		LinkedIn (1 occurrence)	*e.g.,* GET /people/~				
		Musicgraph (8 occurrences)	*e.g.,* GET /api/v2/artist/{id}/albums				
		Twitter (10 occurrences)	*e.g.,* GET /statuses/user_timeline.json				
		StackExchange (53 occurrences)	*e.g.,* GET /answers/{ids}/questions				
		YouTube (9 occurrences)	*e.g.,* GET /users/show.json				
Average				**12.42s**	**89.78%**	**96.67%**	**91.3%**

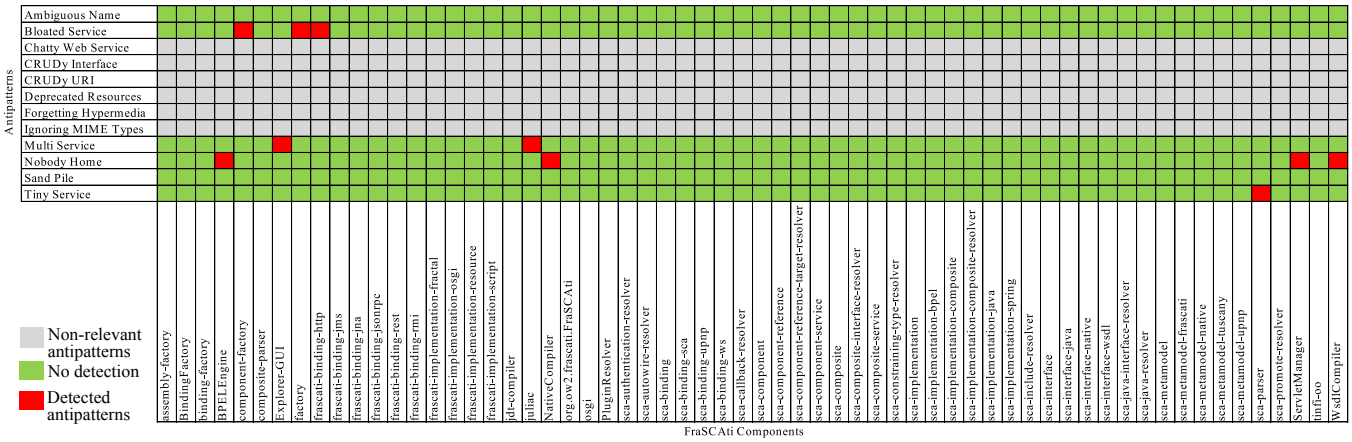


Fig. 15: Detection Results of 12 Antipatterns in the FraSCAti SCA System.

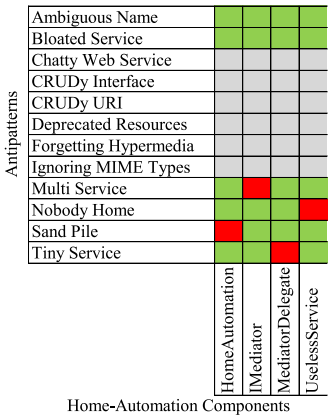


Fig. 16: Detection Results of 12 Antipatterns in the Home-Automation SCA System.

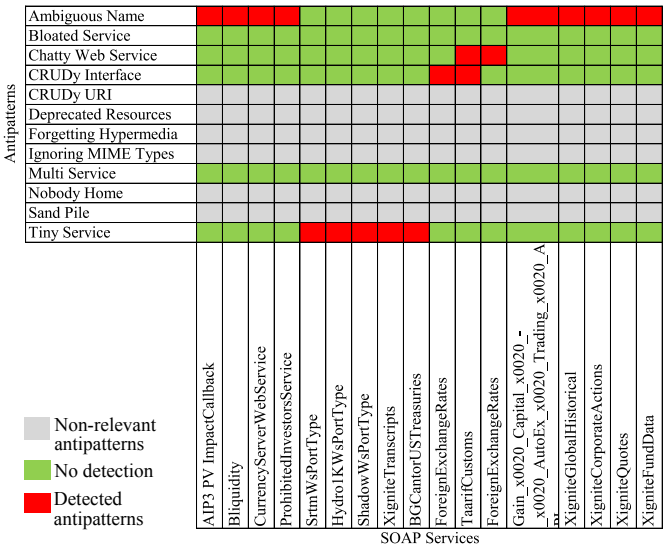


Fig. 17: Detection Results of 12 Antipatterns in 120 SOAP Web Services.

our detection results for each antipattern.

6.5.1 Forgetting Hypermedia and Ignoring MIME Types

RESTful interactions are driven by *hypermedia* with which clients interact through URL links placed in resource representations [2]. In REST, the absence of such *strictly recommended* interaction pattern is commonly known as the *Forgetting Hypermedia* antipattern [44], which we detected in nine RESTful APIs, *e.g.*, Bitly, DropBox, and Facebook (see Table 8 for the full list of APIs). Among the 115 REST requests that we sent, a total of 55 were detected as occurrences of the *Forgetting Hypermedia* antipatterns—REST APIs developers in practice may not always provide links in a resource representation. Out of the 55 detected occurrences, 53 instances were confirmed after manual inspection of all response bodies and headers.

Thus, two occurrences of *Forgetting Hypermedia* are not validated, *i.e.*, our approach identified them as false positives. For example, a condition for a REST API to be an instance of *Forgetting Hypermedia* is the absence of links in the response header or body. In a particular case, using the URL `api.dropboxapi.com/1/metadata/auto/<path>`, we send an HTTP request to *DropBox* to get the meta-data on a file. It returns the HTTP response using the keyword 'path' in its body. However, in our rule card for *Forgetting Hypermedia* antipattern, we compute the `TLB` and `HL` metrics by counting links or locations in the response body and response header, respectively. While calculating those metrics, we only rely on keywords like 'location' or 'link', for which two occurrences are missed where the resource locations are identified using the keyword 'path'. The manual validation rightly considered this detection not to be an instance of *Forgetting Hypermedia* antipattern. Therefore, we plan to extend our framework with other possible synonyms as well as using WordNet for identifying URLs from response body/header in the REST domain. This would improve our precision for *Forgetting Hypermedia* antipattern.

Thus, two occurrences of *Forgetting Hypermedia* are not validated, *i.e.*, our approach identified them as false positives. For example, a condition for a REST API to be an instance of *Forgetting Hypermedia* is the absence of links in the response header or body. In a particular case, using the URL `api.dropboxapi.com/1/metadata/auto/<path>`, we send an HTTP request to DropBox to get the meta-data on a

file. It returns the HTTP response using the keyword *path* in its body. However, in our rule card for *Forgetting Hypermedia* antipattern, we compute TLB and HL metrics counting links or locations in the response body and response header, respectively. To calculate those metrics, we only rely on keywords like *location* or *link*, for which two occurrences are missed where the resource locations are identified using the keyword *path*. The manual validation rightly considered this detection not to be an occurrence of the *Forgetting Hypermedia* antipattern. We plan to extend with the other possible synonyms for identifying URLs from response body/header in the REST domain. This would improve our precision for *Forgetting Hypermedia* antipattern.

We also detected *Ignoring MIME Types* in eight REST services. According to REST principles [2], the server should represent resources in multiple formats, which allow clients a more flexible service consumption. Yet, server-side developers often provide a single representation (or rely on their own formats), which limits the use of resources and service accessibility and reusability. We detected ten instances of *Ignoring MIME Types* antipattern in Twitter and nine instances in YouTube. Moreover, we found in BestBuy and Facebook, only three instances of *Ignoring MIME Types* antipattern, *i.e.*, they mostly follow the good design principle of multiple resource representations.

6.5.2 Multi Service and Tiny Service

From Table 8, we briefly discuss detection results of *Tiny Service* and *Multi Service*. In particular, we detected the IMediator SCA component as a *Multi Service* antipattern in *Home-Automation* due to its very high number of interface methods (NMD or NOD = 13) with a very low cohesion among its methods (COH=0.027) and a very high response time (RT=132ms). All these values are assessed as high or low by the ***Boxplot*** component of SOFA, *i.e.*, compared to the values from other components in *Home-Automation*. For example, the ***Boxplot*** component estimated the median value of NMD (or NOD) in *Home-Automation* as 2, compared to which 13 is quite high. Similarly, the detected *Tiny Service* antipattern, *i.e.*, MediatorDelegate, has a very low number of methods (NMD=l) with a high coupling (CPL=0.44) with respect to other *Home-Automation* components. The cohesion (COH) and coupling (CPL) metrics range between 0 and 1.

We also detected juliac as an instance of the *Multi Service* antipattern in *FraSCAti* due to its high response time (RT=1,018ms), low cohesion (COH=0.1), and high number of methods declared in its interface (NMD=5) when compared to other components. In fact, the median values calculated by the ***Boxplot*** component are: 4ms, 0.1, and 1, respectively. We validated this detection with the core FraSCAti development team, who confirmed that juliac is a *Multi Service* because it implements six different features of two dissimilar abstractions. Furthermore, juliac requires extra execution time because it instantiates the initial SCA system configuration files. Thus, human inspection justifies our detection of juliac as *Multi Service*.

SOFA cannot detect Explorer-GUI as an occurrence of the *Multi Service* antipattern because the scenarios that we tested for *FraSCAti* did not include any use of the graphical interface. Yet, the core development team also reported Explorer-GUI as a *Multi Service*. Our scenarios involved 62 *FraSCAti* components out of 91 existing components.

We also detected SrtmWs-PortType, ShadowWs-PortType, and Hydro1KWs-PortType as instances of the *Tiny Service* antipatterns in SOAP services because they possess very low values for NOD (*i.e.*, 2) and COH (*i.e.*, 0.0). As computed by the *Box-Plot* component in SOFA framework, NOD values of 2 are rather low compared to the median of 5.5. Moreover, the COH values are significantly low compared to other SOAP services whose COH values are in the range of 0.216 and 0.443. Such small services implemented as *Tiny Service* often require other services to function, resulting in higher development complexity and reducing their independent usability. A manual validation of the interfaces of the SOAP Web services confirmed the detection of this antipattern only for ShadowWs-PortType and Hydro1KWs-PortType SOAP services. In fact, for the SrtmWs-PortType service, our external experts who manually validated our results agreed that the operations defined in its interface could fulfill an abstraction, and, thus, they did not consider SrtmWs-PortType service as an antipattern. Our SOFA framework is currently unable to capture the abstraction or context defined within a service interface, which we plan to allow in future work. However, we did not detect any occurrence of *Multi Service* in the SOAP services.

Thus, having a Multi or Tiny Service in an SBS, which exhibits extreme design practices, may degrade the design quality and hinder maintenance of an SBS, should be avoided. Our UniDoSA approach can automatically identify such services with extreme design within SBSs and, therefore, can facilitate maintenance.

6.5.3 Nobody Home

We detected NativeCompiler, ServletManager, WsdlCompiler, BPELEngine components from *FraSCAti* as occurrences of the *Nobody Home* antipattern. For example, the implementation of BPELEngine does not support the weaving of sensors and triggers to introspect at runtime [63], which is discarded by default, by the *FraSCAti* design. In SCA, *weaving* is a technique to introspect components at runtime and allow engineers to measure dynamic and structural properties of each component. Moreover, the BPELEngine component had not been invoked in any of our executed scenarios. The FraSCAti core team partially disagreed with our detection and suggested that it could be invoked in other scenarios, *e.g.*, where FraSCAti particularly handles BPEL scripts.

The UselessService component in *Home-Automation* is also identified as an occurrence of the *Nobody Home* antipattern because it was not invoked in any executed scenarios (*i.e.*, NIR=0 and NMI=0) even though the component was orchestrated with other components in *Home-Automation*. The presence of such components or services not in use within a SBS may increase its maintenance cost. In REST, we did not detect occurrences of the corresponding *Deprecated Resources* antipattern because we implemented only a part of entire resources and tested them all. Thus, no unused or untested resources, *i.e., Deprecated Resources* were found in REST validation.

6.5.4 Ambiguous Name

We detected the AIP3_PV_ImpactCallback as reported in Table 8 as an occurrence of the *Ambiguous Name* antipattern. This Web service offers operations with a set of signatures that (1) are extremely long (ALS=0.675), (2) use high number of general terms for naming (RGTS=0.85), (3) contain many messages having verbs (NVMS=26) and (4) having multiple verbs or action names within a single signature (NVOS=7). In comparison to the median values (*e.g.*, median of ALS=0.463, RGTS=0.0, NVMS=6, and NVOS=3) as calculated by the ***Boxplot*** service component, the computed metric values are high. The SOAP services having *Ambiguous Name* antipattern—which represents poor interface elements naming with (1) very short or long identifiers, (2) too general terms as identifiers, and (3) improper use of verbs—are not semantically and syntactically sound interfaces over the Web and, thus, impact the discoverability and the reusability of services. Applying UniDoSA can help service developers in detecting and refactoring *Ambiguous Name* in SBSs.

6.5.5 Chatty Web Service and CRUDy Interface

We detected ForeignExchangeRates and TaarifCustoms as occurrences of the *Chatty Web Service* and *CRUDy Interface* antipatterns because of their low cohesion (COH=0.16 and 0.12, respectively), high average number of accessor operations (50≤ANAO≤72.22), high number of operations (18≤NOD≤24), and high response time (RT>3s), when compared to other SOAP services. The manual validation did not confirm ForeignExchangeRates as a *Chatty Web Service* because the order of operations invocation can be inferred from the service interface. The specification of *CRUDy Interface* includes *Chatty Web Service*. Therefore, the detection of ForeignExchangeRates as a *CRUDy Interface* was not confirmed.

Thus, having a chatty Web service in a SBS, which exhibits low cohesion among its operations and causes high response time, impacts their maintainability (because inferring the order of invocation is difficult and many interactions are required) and the overall performance. UniDoSA can automatically detect such bottleneck services within SBSs and, therefore, facilitate maintenance.

6.6 Discussion on the Conjectures

We now discuss and support the four conjectures stated in Section 6.1.

6.6.1 Generality

We specified three SCA antipatterns (as listed in Table 2). These antipatterns range from simple ones, such as the *Tiny Service* and *Multi Service*, to complex ones, such as the *Sand Pile*, which involve several services and complex relationships. In particular, *Sand Pile* uses both the ASSOC and COMPOS relation types in its rule card. Also, *Sand Pile* refers, in its specification, to another antipattern, *i.e.*, *DataService*.

As for the SOAP antipatterns, we specified three antipatterns from the literature (see Table 2) that are complex antipatterns with composite rules, such as *CRUDy Interface* composed of another rule card, *i.e.*, *Chatty Web Service*. We also specified antipatterns combining six different rules, *Ambiguous Name* antipattern, for instance.

Using our DSL, derived from our proposed unified meta-model, the generality of defined rule cards for REST antipatterns is also fulfilled because engineers can define rule cards relying on REST-specific metrics and their own experience and knowledge on REST. The proposed DSL allows engineers to specify antipatterns related to HTTP responses (*e.g.*, *Forgetting Hypermedia* and *Ignoring MIME Types*) and related to resource URI design (*e.g.*, *CRUDy URI*).

Thus, we showed that using our unified meta-model and DSL, it is possible to specify from simple to complex antipatterns regardless of the SBSs technology, in particular, for SOAP, SCA, and REST), which supports the generality of our DSL.

6.6.2 Accuracy

Concerning the accuracy of our detection algorithms, as shown in Table 8, we obtained an average recall of 96.67%, precision of 89.78%, and an average F_1-measure of 91.3%. The achievement of such detection accuracy is possible because we defined the rule cards after a thorough literature review and a careful analyses of relevant properties for all service antipatterns and based on a unified meta-model of the service technologies. Also, we automatically generated the detection algorithms using the rule cards.

Thus, we can positively support our second conjecture on the accuracy of our automatically generated (or implemented) detection algorithms.

6.6.3 Extensibility

Our defined DSL is flexible in the integration of new service metrics and antipatterns. For example, we implemented 14 metrics, *e.g.*, AC, AK, CCV, HL, HM, RRF, SC, TLB, and so on as listed in Table 5 dedicated to REST services.

For specifying SCA antipatterns described in the literature (see Table 2), we included metrics dedicated to SCA: Table 4 provides the list of service metrics among which the metrics specific to SCA, namely COH, CPL, NIR, NOR, NUM were initially defined and extracted from the literature.

Later, for specifying new SOAP antipatterns, we reused several pre-existing metrics, for example ANP, ANPT, COH, CPL. We also added some new SOAP metrics to our metric suite, namely ALS, ARIP, NOPT, RGTS, and so on, which are very specific to SOAP services.

However, the underlying SOFA framework should also be extended to provide the operational implementations of the new service metrics. Such an addition can only be realised by developers skilled with our framework, which may require hours according to the complexity of the metrics. However, once the metrics are integrated in the SOFA framework, their use is straightforward for the specification of rule cards using the DSL because the underlying meta-model already encompasses all the elements needed to describe services in the three service technologies.

We added SCA antipatterns (*e.g.*, *Sand Pile*, *Bloated Service*, and *Multi Service*, and so on) within our detection framework first and then extended the framework with new SOAP antipatterns (*e.g.*, *CRUDy Interface*) and REST (*e.g.*, *Forgetting Hypermedia*, *Ignoring MIME Types*), which further confirms the extensibility of our SOFA framework.

Furthermore, our proposed meta-model can be extended with new SBS technologies. However, for this paper it was not required because it already encompasses the three main SBS technologies, namely SCA, SOAP, and REST. However, as a future work, we plan to integrate new SBS technologies to extend our proposed meta-model.

Thus, with these extensibility features of our DSL and our SOFA framework, we positively support A3.

6.6.4 Performance

We performed all our experiments on an Intel Dual Core at 3.30GHz with 4GB of RAM.

For the detection of antipatterns in REST APIs, the total required time includes: (1) the execution time, *i.e.*, sending REST requests and receiving REST responses and (2) the time required to apply and run the detection algorithms on the requests and responses.

For SCA, computation times include introspection time during static and dynamic analyses, computing metric values, and applying detection algorithms.

For SOAP, for each antipattern, the detection time also includes: (1) the filtering of the WSDL files, (2) the generation of the concrete services implementation, (3) the generation of the detection algorithms, and (4) the computation of the related metrics.

Regarding the metrics computation and total detection time, the antipattern computation time is only a fraction of the total detection time as reported in Table 8. The detection time in Table 8 also includes the time required to invoke a service over the Internet, which may significantly vary depending on the bandwidth/traffic. However, the antipatterns computation times, in general, do not vary on a large scale since the detection is performed offline after metrics are computed on-the-fly and range between 10ms and 250ms (with an average of 80ms), which is on average less than 1% of the total detection time reported in Table 8.

As shown in Table 8, the average detection time for all antipatterns (regardless of the technologies) is 12.42s with a minimum of 0.606s (*Nobody Home*) and a maximum of 78.67s (*Multi Service*). Thus, with such a low average detection time of 12.42s, we can positively support our fourth conjecture on performance.

6.7 Discussion

Based on our proposed unified meta-model (in Section 3), we built the UniDoSA approach for the detection of service antipatterns in three service technologies involving three main steps from the specification of service antipatterns to their detection via the automatic generation of detection algorithms (in Section 4). We also listed various service metrics that we used to specify service antipatterns (in Section 4.1). Using those metrics, we defined rule cards for 12 service antipatterns. We also presented the SOFA detection framework (in Section 4.3) and discussed the detailed evaluation of our approach through experimental studies with three commonly used service technologies, *i.e.*, REST, SCA, and SOAP (in Section 6). We stated and positively supported four conjectures on the ***Generality*** of our DSL, the ***Accuracy*** of our detection algorithms, the ***Extensibility*** of our DSL and SOFA framework, and the ***Performance*** of our detection algorithms in Section 6.6. Therefore, we can conclude that:

"*With our proposed UniDoSA approach that encompasses a unified meta-model and a DSL, we can effectively specify and detect service antipatterns in three major service technologies with high accuracy and low performance overhead*."

6.8 Threats to Validity

The main threat to the validity of our results concerns their *external validity*, *i.e.*, the possibility to generalise our approach to other REST services, SCA systems, and SOAP Web services. In this paper, we minimise threats to external validity by performing experiments with 18 common and widely-used REST services, invoking more than 115 methods. We also used one demo SCA system and one large scale SCA system, *i.e.*, *FraSCAti* with more than 150 services. For SOAP Web services, we experimented with more than 120 SOAP services.

For *internal validity*, the detection results not only depend on the services provided by the SOFA framework, but also on the antipattern specifications using rule cards defined for the service antipatterns. To minimise the threat to the internal validity: (1) we unified the elements of the three main service technologies into a unified meta-model, (2) we thoroughly studied the descriptions and definitions of each antipattern from the literature, and (3) we ensured that the SOFA framework does not introduce any antipatterns. Also, we performed experiments on a representative set of antipatterns.

The subjective nature of specifying and validating antipatterns is a threat to *construct validity*. We tried to lessen this threat by defining rule cards for service antipatterns based on a literature review and thorough domain analysis and by involving independent engineers in the validation.

We also tried to minimise the threat to *reliability validity* by automating the generation of the detection algorithms for REST, SCA, and SOAP, such that each subsequent detection produces consistent sets of results with high precision and recall. To further ensure the reliability validity we also used external people (*e.g.*, graduate/undergraduate students, FraSCAti developers, professional engineers) to assess the detection accuracy.

Statistical validity is the extent to which the conclusions drawn from experiments are accurate and reliable. To increase the statistical validity of our results: (1) for REST, we chose multiple providers among the largest REST API providers in the market including *DropBox*, *Facebook*, *Instagram*, *StackExchange*, *Twitter*, *YouTube*, and so on; (2) for SCA, we chose the largest available open-source SCA system, OW2 FraSCAti[5] and an independent, demo SCA application *Home Automation*; (3) for SOAP, we randomly chose in https://www.programmableweb.com more than 120 SOAP services. Our high average precision and recall (89.78% and 96.67%, respectively) on sample sets of services after manual validation suggest that our UniDoSA approach could perform with high accuracy on other SBSs.

5. https://projects.ow2.org/bin/view/frascati/

6.9 Online Tool Support

We have developed a Web-based tool, WEBRESTPAD, to facilitate the analysis of RESTful APIs with the aim of detecting REST patterns and antipatterns in them. The tool is available to use on http://webrestpad.sofa.uqam.ca/.

The main purpose of the WEBRESTPAD tool is the detection of the REST patterns and antipatterns in public RESTful APIs. For a given RESTful API, the tool runs the detection algorithms and displays on the screen for each resource of the API under analysis (1) the detected REST patterns and antipatterns and (2) the traces of each detection showing what makes it a pattern or antipattern.

The key functionalities of the WEBRESTPAD include (1) the addition of a new RESTful API through a new API can be added to the repository that contains all the RESTful APIs under analysis; (2) the detection of REST (anti)patterns that require the invocation of REST services, *a.k.a.*, dynamic REST (anti)patterns, which currently lists 14 REST (anti)patterns; and (3) the detection of REST (anti)patterns that do not require the invocation of REST services, *a.k.a.*, static REST (anti)patterns, which currently lists seven REST (anti)patterns.

Concerning its architecture, the WEBRESTPAD tool is a multi-tier application that consists of a Java EE Web application and a RESTful API. For the dynamic REST (anti)patterns detection, all the functionalities that are available in the Web application are also available in the provided RESTful API.

7 RELATED WORK

Design of quality is essential for building easily maintainable and evolvable SBSs. Service antipatterns are a means to measure design quality.

There are numerous approaches dealing with the detection of OO antipatterns. However, unlike research in the OO domain, research on methods and techniques for the detection of service antipatterns is in its infancy. We performed a thorough literature review following the guidelines by Kitchenham [66] and retrieved relevant research works related to the detection of service-antipatterns and provide a summary.

In fact, the goal of our thorough literature review is to provide a general summary on the detection of antipatterns from all the domains to show what works have been done related to poor design practices (*i.e.*, antipatterns) for object-oriented and service-oriented systems.

We build a search string based on three concepts:

1) *Service oriented architecture* or *object oriented design* (relevant keywords – SOA, SOA architecture, SOA service, service oriented, software architecture, OO, OOD, object oriented);
2) *Antipatterns* (relevant keywords – design pattern, software pattern, software design, pattern detection, antipattern); and,
3) *Service-based systems* or *object oriented systems* (relevant keywords – SBS, SBSs, application software, web service, service base, web application, web service, REST, RESTful, REST service, OO systems, SCA, service component architecture).

With all the search codes and related constraints, our search string is as follows:

"((((SOA* *or* SOA architec* *or* SOA service* *or* service orien* *or* software architec* *or* OO* *or* OOD* *or* object oriented*)wn AB **AND** (design pattern* *or* software pattern* *or* software design* *or* pattern detection* *or* antipattern*)wn TI **AND** (SBS* *or* SBSs* *or* application software* *or* web service* *or* service-base* *or* web application* *or* web service* *or* REST* *or* RESTful* *or* REST service* *or* OO systems* *or* SCA* *or* service component architecture*)wn AB)) **AND** ((({computer software} *or* {software design} *or* {software architecture} *or* {software engineering} *or* {object oriented programming} *or* {web services}) WN CV) *and* (({ca} *or* {ja} *or* {cp}) WN DT) *and* ({english} WN LA)))"

In Table 9, we list works that deal with antipatterns in OO or services in the first column. We also show the types of antipatterns that they consider in the third column; the techniques on which they rely in the fourth column. The fifth and sixth columns show the types and levels of analysis, respectively. The last column shows if those works automated the detection of antipatterns.

In the following sections, based on the results retrieved using our search string defined above, we answer two research questions and summarise relevant research studies in the literature.

7.1 What are the Research Studies Performed in OO Domain on the Detection of Antipatterns?

Table 9 highlights notable works, *e.g.*, [10], [11], [12], [13], [14], [15], [16], [17], [18], in the OO domain inspiring our work on the detection of service antipatterns in SBSs.

OO detection techniques cannot be directly applied to service technologies. Indeed, service technologies have *services* as first-class entities, whereas OO have *classes*, which are at a lower level of granularity. Moreover, the dynamic nature of services raises challenges that are not faced in OO development and requires more dynamic analyses than OO. However, previous works on OO systems form a sound basis of expertise and technical knowledge for building methods for the detection of service antipatterns.

7.2 What are the Research Studies Performed in SO Domain on the Detection of Service Antipatterns?

Various works were carried out for the detection of technology-specific antipatterns, for example, in RESTful APIs [32], in RESTful APIs for cloud services [33], [34], in SCA components [67], [68], and in SOAP Web services [9], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30]. In our previous works [31], [35], [5], we discussed some technology-specific contributions from the literature in details. In particular, we highlighted the significant contributions related to the detection of patterns and antipatterns in SCA, *e.g.*, [67], [68]. There are a number of works for discovering bad practices in writing WSDLs [25], [26], [27], [28], [29], [30]. Some works contribute to the catalog of service antipatterns by defining new service antipatterns, but do not focus on their specification and detection, for example [9], [23], [19], [24]. However, to get the full benefit of those newly defined antipatterns, there should be a generic way to specify and detect them within SBSs regardless of their underlying technologies.

TABLE 9: Relevant Works in the Literature on the Detection of OO, Service, and Cloud Service Antipatterns.

Contributions	Target Systems	Types of Antipatterns	Techniques Used	Type of Analysis	Level of Analysis	Performed Detection?
Salehie *et al.* [10]	OO systems	design-related antipatterns	metrics and rule-based heuristics	design-time	class	✓
Moha *et al.* [11]	OO systems	design-related antipatterns	rule cards, MDE	design-time	class	✓
Stoianov and Sora [12]	OO systems	design-related antipatterns	rules-based (Prolog)	both	class interaction	✓
Smith and Williams [13], [14]	OO systems	performance antipatterns	*n/a*	*n/a*	*n/a*	×
Cortellessa *et al.* [15], [16]	OO systems	performance antipatterns	logical predicates, MDE	design-time	architectural, class	✓
Di Marco and Trubiani [17]	OO systems	performance antipatterns	logical predicates, MDE	run-time	class	✓
Peiris and Hill [18]	OO systems	performance antipatterns	SVM, performance metrics	run-time	class	✓
Palma *et al.* [5]	SCA systems	antipatterns	rule cards, MDE	both	components interaction	✓
Demange *et al.* [67]	SCA systems	design patterns	rule cards, MDE	both	components interaction	✓
Nayrolles *et al.* [68]	SCA systems	design-related antipatterns	mining association rules	run-time	components interaction	✓
Král and Žemlička [9], [23]	Web services	design-related antipatterns	*n/a*	*n/a*	*n/a*	×
Král and Žemlička [19]	Web services	design-related antipatterns	*n/a*	*n/a*	*n/a*	×
Tripathi *et al.* [24]	Web services	design-related antipatterns	*n/a*	*n/a*	*n/a*	×
Torkamani and Bagheri [25]	Web services	design-related antipatterns	*n/a*	*n/a*	*n/a*	×
Anchuri *et al.* [26]	Web services	hotspot services	*n/a*	*n/a*	*n/a*	×
Zheng and Krause [27]	Web services	interaction-related antipatterns	*n/a*	*n/a*	*n/a*	×
Rodriguez *et al.* [28]	Web services	service discoverability antipatterns	information retrieval	design-time	interface	✓
Mateos *et al.* [29]	Web services	WSDL writing antipatterns	information retrieval	design-time	interface	✓
Rodriguez *et al.* [30]	Web services	WSDL writing antipatterns	rule-based technique	design-time	interface	✓
Palma *et al.* [35]	Web services	service design and documentation	rule cards, MDE	both	interface	✓
Ouni *et al.* [20]	Web services	WSDL writing antipatterns	search-based, rule-based	design-time	interface	✓
Wang *et al.* [22]	Web services	WSDL writing antipatterns	bi-level optimization, rule-based	design-time	interface	✓
Ouni *et al.* [21]	Web services	WSDL writing antipatterns	genetic programming, rule-based	design-time	interface	✓
Palma *et al.* [31]	REST APIs	syntactic design of requests/responses	heuristics-based	run-time	resource	✓
Rodríguez *et al.* [32]	REST APIs	HTTP requests	heuristics-based	run-time	resource	✓
Petrillo *et al.* [33]	REST APIs for Cloud	HTTP requests, URI design	ontology-based, rule-based	run-time	interface	✓
Brabra *et al.* [34]	REST APIs for Cloud	HTTP requests, URI design	heuristics-based	run-time	interface	✓

7.3 Maturity Model vs. Antipatterns

The *Richardson Maturity Model* [69] estimates how 'RESTful' Web services are. In the four-layered model, Richardson proposed to use three factors (*e.g.*, URI, HTTP methods, and hypermedia) to decide the maturity of a service. The more a service employs these three factors, the more mature it is. This model measures how much a REST service is reaping the benefits of REST, for example, the use of tunneling, multiple resources, HTTP verbs, and hypermedia controls.

Instead of verifying the RESTfulness of Web services, our approach focuses on assessing their design quality. Even if a Web service complies with Richardson Maturity Model, it does not guarantee that the service is well-designed from an architectural point of view and would not hinder the maintenance and evolution of SBSs. Therefore, rather than focusing on RESTfulness, our approach focuses on the design quality of Web services. In addition to that our approach is technology-neutral, *i.e.*, the maturity model applies to REST only while our unified approach is capable of assessing design quality of SBSs regardless their underlying technology.

7.4 Summary

We identify the following gaps in the literature:

- Numerous contributions are presented in the literature for analysing OO design quality to detect antipatterns in OO systems, whereas the analysis of SBSs was exploited a little. Those OO approaches are not directly applicable to SBSs;
- Several approaches were proposed in the literature to analyse SCA systems, relying on the analysis of execution traces [68] or metric-based quantitative analysis [67]. However, these approaches are proposed considering the SCA systems only;
- A number of empirical validations were performed in the literature on the detection of antipatterns in SOAP services interfaces, *i.e.*, discovering bad practices in writing WSDL or identifying best practices [28], [30], [24], [20], [21], [22]. However, these analyses are static and do not incorporate runtime aspects (*e.g.*, *availability* or *response time*) of SOAP services;
- For REST, the works in [31], [32] existed dealing with antipatterns in REST services, which was unable to handle SCA or SOAP services. A more rigorous assessment of REST services is required. Moreover, some works have been done on detecting antipatterns in RESTful APIs for cloud services [33], [34];
- There exists no unified meta-model that combines different service technologies and provide a means to detect service antipatterns in SBSs developed using diverse technologies in a generic way;
- There exists no unified domain-specific language (DSL) that can facilitate the representation of service antipatterns without ambiguity regardless of SBS technologies. This unified DSL is only possible once we define a unified meta-model.
- A unified framework-based approach is missing in the literature for the static/dynamic analyses of SBSs.

We filled the above gaps in the literature and contributed with a unified detection approach by proposing a unified meta-model that describe three major service technologies, building a domain-specific language to describe antipatterns, and relying on an underlying detection framework that can support the specification of service antipatterns.

8 Conclusion

Service-based Systems (SBSs), relying on *services* as first-class entities, are developed on top of diverse service technologies and architectural styles. The REST, SCA, and SOAP Web Service technologies are widely used by companies to design and develop SBSs [2], [3], [4].

SBSs are subject to *functional* and *non-functional* changes, which may degrade their design and implementation and introduce *service antipatterns*. Antipatterns in SBSs (1) may hinder their further maintenance and evolution and (2) may degrade their design quality and quality of service (QoS). Such antipatterns must be detected to improve their design and QoS, and ease their maintenance and evolution.

Among the many reasons for the antipatterns to occur, *ignorance* of developers and *time to deliver* constraint are notable. For example, *Multi Service* antipattern may easily occur when a developer tries to put too much responsibilities under a single service without knowing the consequences, and, in an opposite scenario, a *Tiny Service* is very easy to introduce by developing services with one or very few fine-grained tasks without a complete abstraction. However, finding a proper balance between a *Multi Service* and a *Tiny Service*, *i.e.*, to form the proper abstraction, is always a challenge for service developers. Another way of introducing antipatterns by the designers is selecting an improper set of services or orchestrating services in an improper manner. This may meet the functional requirements without fulfilling the non-functional goals, which in turn can degrade the system performance due to poor design choices. Our goal is to provide a unique solution for engineers to help them understand the service domain and to get rid of service antipatterns. We do this through the two following aspects.

First, we presented a unified meta-model to support the unified specification of service antipatterns in different service technologies. Then, we proposed an unified approach, UniDoSA (**Uni**fied Specification and **D**etection **o**f **S**ervice **A**ntipatterns) that uses the unified meta-model to assess the design quality and QoS of SBSs by means of the detection of service antipatterns in SBSs. UniDoSA is supported by an underlying framework, SOFA (Service Oriented Framework for Antipatterns), which provides a common platform to: (1) specify service antipatterns at higher-level of abstraction, (2) generate detection algorithms for service antipatterns automatically, (3) apply generated detection algorithms on diverse SBSs and report candidate service antipatterns involved in service antipatterns, and (4) validate the candidate service antipatterns to confirm them as real antipatterns.

We showed the effectiveness of the UniDoSA approach in four steps: (1) ***Generality*** – the domain specific language (DSL) based on the unified meta-model is flexible to define antipatterns rule cards in various service technologies, (2) ***Accuracy*** – our specified rule cards have a high accuracy with average detection precision and recall of more than 75%, *i.e.*, we obtained an average precision of 89.78% and an average recall of 96.67%, (3) ***Extensibility*** – the DSL and the SOFA framework are extensible for adding new service metrics and antipatterns, and (4) ***Performance*** – the generated detection algorithms perform with low overhead, *i.e.*, the average detection times are in the order of seconds. Major challenges on extending the SOFA include: engineers require thorough knowledge on the framework and the addition of new metrics or service technology is time demanding because it is not automatic and requires repeating the first three steps (*i.e.*, specification after domain analysis, generation, and detection) of our UniDoSA approach.

In summary, UniDoSA is the first unified approach to detect service antipatterns in SBSs regardless of their underlying implementation technologies and it provides high detection accuracy.

As future work, we want to verify the impact of service antipatterns on the maintenance of SBSs. It would be interesting to see how the detected antipatterns influence the systems in which they have been detected. Indeed, we plan to see empirically how these systems under study perform having antipatterns in them, compared to, while those detected antipatterns are re-factored by modifying or improving their design. We also plan to use OCL rules on our meta-model for two reasons: (1) to check its integrity and (2) to enable the antipatterns navigating the unified meta-model that will facilitate their specification. In the line of future work, we also plan to work on the correction of the detected service antipatterns, in particular by involving developers in semi-automated approaches.

Acknowledgment

The authors are thankful to Dr. Wei Wu for his valuable insights and comments on this paper. We would like to thank Moustapha Boulgoudan from University of Québec in Montréal, Canada and GaÃńtan FranÃğois from Polytech Montpellier, France for their help with the implementation of the online antipattern detection tool. This work was partly supported by NSERC, Canada Chairs, and FRQ-NT research grants.

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, August 2005.

[2] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, 2000.

[3] D. Chappell, *Introducing SCA*. USA: Chappell & Associates, July 2007.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, "Web Services: Concepts, Architectures and Applications," ser. Data-Centric Systems and Applications. Springer, 2003.

[5] F. Palma, M. Nayrolles, N. Moha, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "SOA Antipatterns: An Approach for their Specification and Detection," *International Journal of Cooperative Information Systems*, vol. 22, no. 04, 2013.

[6] A. Koenig, "Patterns and antipatterns," *The patterns handbook: techniques, strategies, and applications*, pp. 383–390, 1998.

[7] W. J. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.

[8] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf, *J2EE AntiPatterns*. John Wiley and Sons, August 2003.

[9] J. Král and M. Žemlička, "Crucial Service-Oriented Antipatterns," in *In Proceedings of the International Conference on Software Engineering Advances*, vol. 2, no. 1. International Academy, Research and Industry Association (IARIA), 2008, pp. 160–171.

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

[11] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transaction on Software Engineering*, vol. 36, no. 1, pp. 20–36, January 2010.

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

[13] C. U. Smith and L. G. Williams, "New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot," in *International Computer Measurement Group Conference*, 2002, pp. 667–674.

[14] C. U. Smith and L. G. Williams, "Software Performance Antipatterns," in *Proceedings of the 2nd International Workshop on Software and Performance*, ser. WOSP '00. New York, NY, USA: ACM, 2000, pp. 127–136.

[15] V. Cortellessa, A. Di Marco, and C. Trubiani, "Software Performance Antipatterns: Modeling and Analysis," in *Formal Methods for Model-Driven Engineering*, ser. Lecture Notes in Computer Science, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer Berlin Heidelberg, 2012, vol. 7320, pp. 290–335.

[16] V. Cortellessa, A. Di Marco, and C. Trubiani, "An Approach for Modeling and Detecting Software Performance Antipatterns based on First-order Logics," *Software & Systems Modeling*, vol. 13, no. 1, pp. 391–432, 2014.

[17] A. D. Marco and C. Trubiani, "A model-driven approach to broaden the detection of software performance antipatterns at runtime," in *Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, Grenoble, France*, 2014, pp. 77–92.

[18] M. Peiris and J. H. Hill, "Towards Detecting Software Performance Anti-patterns Using Classification Techniques," *SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–4, Feb. 2014.

[19] J. Král and M. Žemlička, "Popular SOA Antipatterns," in *Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, ser. COMPUTATIONWORLD '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 271–276.

[20] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinnéide, "Search-based Web Service Antipatterns Detection," *IEEE Transaction on Service Computing*, vol. 01, 2015.

[21] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, "Web Service Antipatterns Detection Using Genetic Programming," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '15. New York, NY, USA: ACM, 2015, pp. 1351–1358.

[22] H. Wang, M. Kessentini, and A. Ouni, *Bi-level Identification of Web Service Defects*. Cham: Springer International Publishing, 2016, pp. 352–368.

[23] J. Král and M. Žemlička, "The Most Important Service-Oriented Antipatterns," in *International Conference on Software Engineering Advances*, Aug 2007, pp. 29–29.

[24] D. Tripathi, U. Suman, M. Ingle, and S. K. Tanwani, "Towards Introducing and Implementation of SOA Design Antipatterns," *International Journal of Computer Theory and Engineering*, vol. 6, no. 1, pp. 20–25, February 2014.

[25] M. A. Torkamani and H. Bagheri, "A Systematic Method for Identification of Antipatterns in Service Oriented System Development," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 4, no. 1, pp. 16–23, 2014.

[26] P. Anchuri, R. Sumbaly, and S. Shah, "Hotspot Detection in a Service-Oriented Architecture," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, ser. CIKM '14. New York, NY, USA: ACM, 2014, pp. 1749–1758.

[27] Y. Zheng and P. Krause, "Asynchronous Semantics and Antipatterns for Interacting Web Services," in *6th International Conference on Quality Software (QSIC)*, Oct 2006, pp. 74–84.

[28] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Improving Web Service Descriptions for Effective Service Discovery," *Science of Computer Programming*, vol. 75, no. 11, pp. 1001–1021, 2010.

[29] C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, "Detecting WSDL Bad Practices in Code-first Web Services," *International Journal of Web and Grid Services*, vol. 7, no. 4, pp. 357–387, Jan. 2011.

[30] J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, "Best Practices for Describing, Consuming, and Discovering Web Services: A Comprehensive Toolset," *Software: Practice and Experience*, vol. 43, no. 6, pp. 613–639, 2013.

[31] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, X. Franch, A. Ghose, G. Lewis, and S. Bhiri, Eds., vol. 8831. Springer Berlin Heidelberg, 2014, pp. 230–244. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-45391-9_16

[32] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, *REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices*. Cham: Springer International Publishing, 2016, pp. 21–39.

[33] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, *Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study*. Cham: Springer International Publishing, 2016, pp. 157–170.

[34] H. Brabra, A. Mtibaa, L. Sliman, W. Gaaloul, B. Benatallah, and F. Gargouri, *Detecting Cloud (Anti)Patterns: OCCI Perspective*. Cham: Springer International Publishing, 2016, pp. 202–218.

[35] F. Palma, N. Moha, G. Tremblay, and Y.-G. GuÃľhÃľneuc, "Specification and Detection of SOA Antipatterns in Web Services," in *Software Architecture*, ser. Lecture Notes in Computer Science, P. Avgeriou and U. Zdun, Eds., vol. 8627. Springer International Publishing, 2014, pp. 58–73. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09970-5_6

[36] FraSCAti, "Home-Automation," June 2013. [Online]. Available: websvn.ow2.org/listing.php?repname=frascati&path=/trunk/demo/home-automation/

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

[38] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, March 2005.

[39] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Automatically Detecting Opportunities for Web Service Descriptions Improvement," W. Cellary and E. Estevez, Eds. Springer Berlin Heidelberg, 2010, vol. 341, pp. 139–150.

[40] T. Modi, "SOA Management: SOA Antipatterns," August 2006.

[41] S. Jones, "SOA Anti-patterns, Available Online: www.infoq.com/articles/SOA-anti-patterns," June 2006.

[42] J. Evdemon, "Principles of Service Design: Service Patterns and Anti-Patterns," August 2005.

[43] M. Massé, *REST API Design Rulebook*. O'Reilly, 2012.

[44] S. Tilkov, "REST Anti-Patterns, Available Online: www.infoq.com/articles/rest-anti-patterns," July 2008.

[45] F. Valverde and O. Pastor, "Dealing with REST Services in Model-driven Web Engineering Methods," *V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB*, pp. 243–250, 2009.

[46] W. B. Abid, M. Graiet, M. Kmimech, M. T. Bhiri, W. Gaaloul, and E. Cariou, "Profile UML2.0 for Specification of the SCA Architectures," *Semantics, Knowledge and Grid, International Conference on*, vol. 0, pp. 191–194, 2011.

[47] WWW-Consortium, "WWW Consortium, Web Services Description Language (WSDL) Version 2.0," Tech. Rep., January 2006.

[48] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[49] L. Cherbakov, M. Ibrahim, and J. Ang, "SOA Antipatterns: The Obstacles to the Adoption and Successful Realization of Service-Oriented Architecture," January 2006.

[50] R. Prieto-Díaz, "Domain Analysis: An Introduction," *SIGSOFT Softw. Eng. Notes*, vol. 15, no. 2, pp. 47–54, Apr. 1990.

[51] C. Consel and R. Marlet, "Architecturing Software Using A Methodology for Language Development," *Lecture Notes in Computer Science*, vol. 1490, pp. 170–194, September 1998.

[52] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen, *Generative Programming and Active Libraries*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 25–39. [Online]. Available: https://doi.org/10.1007/3-540-39953-4_3

[53] L. Geiger and et al., "Template- and Model-based Code Generation for MDA-tools," in *Proceedings Of The Fujaba Days 2005, Volume Tr-Ri-05-259 Of Technical Report. University Of Paderborn*, 2005, pp. 57–62.

[54] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, "Automatic Code Generation from Design Patterns," *IBM Systems Journal*, vol. 35, no. 2, pp. 151–171, 1996.
[55] EMF-Eclipse, "Eclipse Modeling Framework (EMF) - http://www.eclipse.org/emf," April 2010.
[56] D. Sciamma, G. Cannenterre, and J. Lescot, "Ecore Tools," www.eclipse.org/modeling/emft/?project=ecoretools, Tech. Rep., May 2013.
[57] EMFText, "http://www.emftext.org/," www.eclipse.org/acceleo, Tech. Rep., 2007.
[58] Obeo, "Acceleo," www.eclipse.org/acceleo, Tech. Rep., 2005.
[59] G. A. Miller, "WordNet: A Lexical Database for English," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/219717.219748
[60] J. Chambers, W. Cleveland, P. Tukey, and B. Kleiner, *Graphical Methods for Data Analysis*. Wadsworth International, 1983.
[61] M. Perepletchikov, C. Ryan, and Z. Tari, "The Impact of Service Cohesion on the Analyzability of Service-Oriented Software," *IEEE Transcations on Services Computing*, vol. 3, no. 2, pp. 89–103, April 2010.
[62] F. Palma, N. Moha, , and Y.-G. Guéhéneuc, "A Technical Report on UniDoSA (The Unified Specification and Detection of Service Antipatterns)," Tech. Rep., January 2018. [Online]. Available: http://sofa.uqam.ca/media/unidosa-tr01.pdf
[63] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures," *Software: Practice and Experience*, vol. 42, no. 5, pp. 559–583, May 2012.
[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
[65] S. Tilkov, "RESTful Design: Intro, Patterns, Anti-Patterns, Available Online: http://www.devoxx.com," December 2008.
[66] B. Kitchenham, "Procedures for performing systematic reviews," July 2004.
[67] A. Demange, N. Moha, and G. Tremblay, "Detection of SOA Patterns," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds., vol. 8274. Springer Berlin Heidelberg, 2013, pp. 114–130.
[68] M. Nayrolles, N. Moha, and P. Valtchev, "Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces," in *20th Working Conference on Reverse Engineering*, October 2013, pp. 321–330.
[69] L. Richardson, "Richardson Maturity Model," Tech. Rep., 2010.

Francis Palma is currently an Assistant Professor at the Department of Computer Science in Linnaeus University, Sweden. Francis earned his PhD in 2015 from Polytechnique Montréal (University of Montreal), Canada under the supervision of Dr. Naouel Moha and Dr. Yann-Gaël Guéhéneuc. His main research interests include analysing the design quality and evaluating the QoS of service-oriented systems, in particular, detecting service patterns/antipatterns in service-based systems.

Naouel Moha is currently Associate Professor at the Department of Informatics at the University of Québec in Montréal and adjunct director of the institutional research centre LATECE (Laboratory for Research on Technology for E-commerce). Her research works focus on software quality, maintenance and evolution. In particular, she is interested in the detection of patterns and antipatterns in object and service-oriented systems. She received a PhD from the University of Montreal (Canada) and the University of Lille (France).

Yann-Gaël Guéhéneuc received the PhD degree in software engineering from the University of Nantes, France in 2003, under the supervision of Dr. Pierre Cointe. He is currently a full professor at the Department of Computer Science and Software Engineering of Concordia University, where he leads the Ptidej Team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural levels. His research interests include program understanding and program quality during development and maintenance, in particular through the use and identification of recurring patterns. He has published many papers in international conferences and journals.