International Journal of Cooperative Information Systems
© World Scientific Publishing Company

SOA ANTIPATTERNS: AN APPROACH FOR THEIR SPECIFICATION AND DETECTION

FRANCIS PALMA

Ptidej Team, DGIGL, École Polytechnique de Montréal, C.P. 6079, Succ. Centre-ville Montréal, Quebec, H3C 3A7, Canada
Département d'informatique, Université du Québec à Montréal, C.P. 8888, Succ. Centre-ville Montréal, Quebec, H3C 3P8, Canada

MATHIEU NAYROLLES AND NAOUEL MOHA

Département d'informatique, Université du Québec à Montréal, C.P. 8888, Succ. Centre-ville Montréal, Quebec, H3C 3P8, Canada

YANN-GAËL GUÉHÉNEUC

Ptidej Team, DGIGL, École Polytechnique de Montréal, C.P. 6079, Succ. Centre-ville Montréal, Quebec, H3C 3A7, Canada

BENOIT BAUDRY AND JEAN-MARC JÉZÉQUEL

INRIA Rennes, Université Rennes 1
Rennes, France

Received (Day Month Year)
Revised (Day Month Year)

Like any other large and complex software systems, Service Based Systems (SBSs) must evolve to fit new user requirements and execution contexts. The changes resulting from the evolution of SBSs may degrade their design and quality of service (QoS) and may often cause the appearance of common poor solutions in their architecture, called *antipatterns*, in opposition to *design patterns*, which are good solutions to recurring problems. Antipatterns resulting from these changes may hinder the future maintenance and evolution of SBSs. The detection of antipatterns is thus crucial to assess the design and QoS of SBSs and facilitate their maintenance and evolution. However, methods and techniques for the detection of antipatterns in SBSs are still in their infancy despite their importance. In this paper, we introduce a novel and innovative approach supported by a framework for specifying and detecting antipatterns in SBSs. Using our approach, we specify ten well-known and common antipatterns, including *Multi Service* and *Tiny Service*, and automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on two systems developed independently, (1) *Home-Automation*, an SBS with 13 services, and (2) *FraSCAti*, an open-source implementation of the Service Component Architecture (SCA) standard with more than 100 services. This validation demonstrates that our approach enables the specification and detection of SOA antipatterns with an average precision of 90% and recall of 97.5%.

Keywords: Antipatterns; Service based systems; Service Component Architecture; Spec-

ification; Detection; Quality of service; Design; Software evolution and maintenance.

1. Introduction

Service Oriented Architecture (SOA) is an emerging architectural style that is becoming broadly adopted in industry because it allows the development of low-cost, flexible, and scalable distributed systems by composing ready-made services, *i.e.*, autonomous, reusable, and platform-independent software units that can be accessed through a network, such as the Internet [1]. This architectural style can be implemented utilizing a wide range of SOA technologies, such as OSGi, SCA, REST, RPC, and Web Services. SOA allows building different types of Service Based Systems (SBSs) from business systems to cloud-based systems. Google Maps, Amazon, eBay, PayPal, and FedEx are examples of SBSs. Spanoudakis and Mahbub defined SBSs as the composite systems, which are dynamically composed with autonomous Web services such that they are controlled by some composition processes [2]. We further generalise the definition:

"*SBSs are systems that are built on top of SOA principles and are composed of services implemented with heterogeneous technologies as their building blocks.*"

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

Multi Service and *Tiny Service* are two common antipatterns in SBSs and it has been shown, in particular, that *Tiny Service* is the root cause of many SOA failures [3]. *Multi Service* is an SOA antipattern that represents a service implementing a multitude of methods related to different business and technical abstractions. Such a service is not easily reusable because of the low cohesion of its methods and is often unavailable to end-users because it is overloaded. Conversely, *Tiny Service* is a small service with just a few methods, which only implements part of an abstraction. Such service often requires several coupled services to be used together, leading to higher development complexity and reduced flexibility. These two antipatterns represent extreme in the balance between cohesion and coupling.

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

their infancy. In this paper, we consider the SCA systems to analyse and to detect antipatterns while keep other SOA technologies as our future work. However, the availability of free SBSs as test bed is a major challenge that we might face.

Our goal is to assess the design and QoS of SBSs. To achieve this goal, we already proposed a novel and innovative approach, SODA (Service Oriented Detection for Antipatterns) supported by a framework, SOFA (Service Oriented Framework for Antipatterns), to specify and detect SOA antipatterns automatically in SBSs [4]. This framework supports the static and dynamic analysis of SBSs, along with their combination. Static analysis involves measurement of structural properties related to the design of SBSs while dynamic analysis requires the runtime execution of SBSs for the measurement of runtime properties, mainly related to QoS.

The SODA approach relies on the first language to specify SOA antipatterns in terms of *metrics*. This language is defined from a thorough domain analysis of SOA antipatterns from the literature. It allows the specifications of SOA antipatterns using high-level domain-related abstractions. It also allows the adaptation of the specifications of antipatterns to the context of the analysed SBSs. Using this language and the SOFA framework dedicated to the static and dynamic analysis of SBSs, we generate detection algorithms automatically from the specifications of SOA antipatterns and apply them on any SBSs under analysis. The originality of our approach stems from the ability for software engineers to specify SOA antipatterns at a high-level of abstraction using a consistent vocabulary and from the use of a domain-specific language for automatically generating the detection algorithms.

In our previous work [4], we assessed the effectiveness of the proposed approach only on a small scale SBS, *i.e.*, *Home-Automation*. The results indicated that we had good precision (92.5%) and recall (100%) on *Home-Automation*, after specifying SOA antipatterns and generating their detection algorithms automatically. In the extension to our previous work, we apply SODA by specifying 10 well-known and common SOA antipatterns and generating their detection algorithms. Then, we validate the detection results in terms of precision and recall on two systems: (1) *Home-Automation*, an SBS developed independently by two Masters students that involves 13 services and (2) *FraSCAti* [5, 6], an open-source implementation of the SCA standard with more than 100 services. *FraSCAti* is almost ten times bigger than *Home-Automation* in terms of the number of constituent services. We also consider two different versions of *Home-Automation*: (a) an original version, which includes 13 services and (b) a version modified by adding and modifying services to inject intentionally some antipatterns. We show that SODA allows the specification and detection of a representative set of SOA antipatterns with an average precision of 90.84% and a recall of 97.5%.

In summary, our contributions from [4] include:

(1) A first approach, SODA, for specifying and detecting SOA antipatterns supported by an underlying framework, SOFA;

(2) An extensible domain specific language (DSL) to specify SOA antipatterns;
(3) A validation of the approach with a small scale SBS, *i.e.*, *Home-Automation*.

This paper extends our previous work [4] with the following additional contribution: an extensive validation of the proposed approach with a large scale SBS, *i.e.*, *FraSCAti* [5,6], which is an open-source implementation of the SCA standard, and the largest SCA system available. We also complement our previous work [4] with a more elaborated and up-to-date related work. In addition, we also specify three new SOA antipatterns and perform their detection on *FraSCAti*. Thus, we extend our domain specific language (DSL) to accommodate new metrics, and extend the repository of detected antipatterns.

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

2. Related Work

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

2.1. *Detection of SOA Patterns*

The current catalog of SOA design patterns is rich enough: there exists a number of books on SOA patterns and principles [7–9] that provide guidelines and principles characterizing "good" service-oriented designs. Such books enable software engineers to manually evaluate the quality of their systems and provide a ground for improving design and implementation. For example, Rotem-Gal-Oz *et al.* [9] introduced 23 SOA patterns and four SOA antipatterns and discussed their consequences, causes, and corrections. Erl, in his book [7], introduced more than 80 SOA design-, implementation-, security-, and governance-related patterns.

However, there are a very few contributions on detecting patterns in the SOA context. One prominent contribution towards detecting SOA patterns is by Upadhyaya *et al.* [10], in which the authors devised an approach based on execution log analysis. The authors detected 9 service-composition related SOA patterns that are mostly domain-specific and goal-oriented: if the application is related to the domain of financial or supply-chain management, then the composition patterns can eas-

ily be observed based on the use-case scenarios. However, the proposed approach cannot assess the quality of an SBS design and its performance in general. There are also a few works on detecting SOA patterns at service-level, *i.e.*, similarities among services [11], and at process-level, *i.e.*, within workflows [12–14] but none of them considered patterns at design-level. Unlike these works, we are interested in patterns and antipatterns, that assess the quality of design and QoS to ease the maintenance and evolution of SBSs.

2.2. *Detection of SOA Antipatterns*

Unlike SOA patterns, fewer books and papers deal with SOA antipatterns: most references are Web sites where SOA practitioners share their experiences in SOA design and development [15–17]. In 2003, Dudney *et al.* [18] published the first book on SOA antipatterns. This book provides a catalog of 53 antipatterns related to the architecture, design, and implementation of systems based on J2EE technologies, such as EJB, JSP, Servlet, and Web Services. Most antipatterns described in this book cannot be detected automatically and are specific to a technology and correspond to variants of the Tiny and Multi Service. Král *et al.* [3] also described seven SOA antipatterns, which are caused by an improper usage of SOA standards and improper practices borrowed from the OO design style.

While the catalog of SOA antipatterns is growing, there are very few dedicated approaches for detecting SOA antipatterns in SBSs, *i.e.*, the approach proposed by Trčka *et al.* [19] and ours in [4]. Trčka *et al.* [19] proposed a technique to discover data-flow antipatterns based on model-checking. They specified using temporal logic nine data-flow antipatterns (*e.g.*, *Missing Data*, *Inconsistent Data*, *Not Deleted on Time*, etc.) and detected them by analysing data dependencies within workflows and improper data handling. In [4], we also proposed a means for the specification and detection of SOA antipatterns that we recall in the following for consistency. Trčka *et al.* focused on data-flow antipatterns whereas we focus on detecting antipatterns to assess and improve the quality of design and QoS of SBSs. However, we observe from the literature that due to the smaller 'catalog size' and limited available resources, *i.e.*, articles, books, journals, etc., the specification and detection of SOA antipatterns were not considered with greater importance by the SOA community.

2.3. *Detection of OO Antipatterns*

In the contrary to the efforts given on detecting SOA patterns and antipatterns, a number of methods and tools exist for the detection of antipatterns in OO systems [20–25] and various books have been published on that topic. For example, Brown *et al.* [26] introduced a collection of 40 antipatterns, Beck, in Fowler's highly-acclaimed book on refactoring [27], compiled 22 code smells that are low-level antipatterns in source code, suggesting where engineers should apply refactorings.

Among the previous works, DECOR [20] is a rule-based approach for the specification and detection of code and design smells in OO systems that can be related to our work. The authors use a domain specific language to specify smells and then automatically generate smell-detection algorithms, which are directly executable. DECOR can detect smells in OO systems with a precision of 60.5% and recall of 100%. Later, Kessentini *et al.* [21] improved the detection precision by automating the rule construction. The authors use *genetic programming* for finding an optimal set of rules to maximize the smell detection. Also, Khomh *et al.* [28] proposed an GQM-based (Goal Question Metric) approach relying on the definition of antipattern rather than the rule card, and improved precision and recall from its state-of-the-art approach, DECOR. However, being inspired from DECOR, we intend to follow a similar approach in the context of the service-oriented paradigm for detecting SOA antipatterns.

Other related works have focused on the detection of specific antipatterns related to system's performance and resource usage and–or given technologies. For example, Wong *et al.* [29] used a genetic algorithm for detecting software faults and anomalous behavior related to the resource usage of a system (*e.g.*, memory usage, processor usage, thread count). Their approach is based on *utility functions*, which correspond to predicates that identify suspicious behavior based on resource usage metrics. For example, a utility function may report an anomalous behavior corresponding to spam sending if it detects a large number of threads. In another relevant work, Parsons *et al.* [30] introduced an approach for the detection of performance antipatterns specifically in component-based enterprise systems (in particular, J2EE applications) using a rule-based approach relying on static and dynamic analyses.

Moreover, there exists tools proposed by the industry and the academic community for automating the detection of OO code smells and antipatterns, such as cbsdetector [31], FindBugs [32], iPlasma [33], JDeodorant [34], PatOMat [35], PMD [36], SonarQube [37], SPARSE [38], etc.

2.4. *Summary*

One of the root causes of OO antipatterns is the adoption of a procedural design style in OO system whereas for SOA antipatterns, it stems from the adoption of an OO style design in SOA system [3]. OO detection methods and tools cannot be directly applied to SOA because SOA focuses on services as first-class entities whereas OO focuses on classes, which are at a lower-level of granularity. Moreover, the highly dynamic nature of an SOA environment raises several challenges that are not faced in OO development and requires more dynamic analyses.

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

3. The SODA Approach

In [4], we proposed a three-step approach, named SODA, for the specification and detection of SOA antipatterns as shown in Figure 1. We recall this approach:
Step 1. Specify SOA antipatterns: This step lies in identifying properties in SBSs relevant to SOA antipatterns. These properties can also be referred to as *metrics*. Using those properties, we define a Domain-Specific Language (DSL) for specifying antipatterns at a high level of abstraction.
Step 2. Generate detection algorithms: In this step, detection algorithms are generated automatically from the specifications defined in the previous step.
Step 3. Detect automatically SOA antipatterns: The third step consists of applying, on the SBSs of interest, the detection algorithms generated in *Step 2* to detect SOA antipatterns.

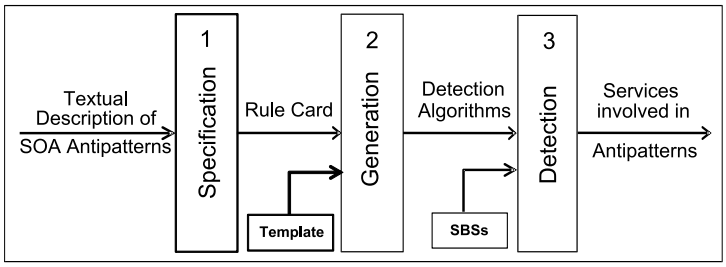

Figure 1. The SODA Approach.

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

3.1. *Specification of SOA Antipatterns*

As a prerequisite to specify antipatterns, we perform a thorough domain analysis of SOA antipatterns by studying their definitions and specifications in the literature [3,9,18] and in online resources and articles [15–17]. This domain analysis allows us to identify properties relevant to SOA antipatterns, including static properties related to their design (*e.g.*, cohesion and coupling) and also dynamic properties, such as QoS criteria (*e.g.*, response time and availability). Static properties are properties that apply to the static descriptions of SBSs, such as WSDL (Web Services Description Language, for Web Services) and SCDL (Service Component Definition Language, for SCA) files, whereas dynamic properties are related to the dynamic behavior of SBSs as observed during their execution. We use these properties as a base vocabulary to define a DSL, in the form of a rule-based language for specifying SOA antipatterns. The DSL offers software engineers with high-level domain-related abstractions and variability points to express different properties of antipatterns depending on their own judgment and context.

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

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

 5  ruleType      ::= ruleName | rule_cardName

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

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

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

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

Figure 2. BNF Grammar of Rule Cards.

We specify antipatterns using rule cards, *i.e.*, sets of rules. We formalize rule cards with a Backus-Naur Form (BNF) grammar, which determines the syntax of our DSL. Figure 2 shows an extended version of the grammar that we first proposed in [4] and used to express the rule cards. A rule card is identified by the keyword `RULE_CARD`, followed by a name and a set of rules specifying this specific antipattern (Figure 2, line 1). A rule (line 3 and 4) describes a metric, an association or composition relationship among rules (lines 14-16) or a combination with other rules, based on set operators including intersection, union, difference, inclusion, and negation (line 6). Each rule returns a set of services. Therefore, it is possible to apply a set operator between at least two rules and obtain the result of this set operation again in the form of a set of services. A rule can refer also to another rule card previously specified (line 4). A metric associates to an identifier a numerical or an ordinal value (lines 7 and 8). Ordinal values are defined with a five-point Likert scale: very high, high, medium, low, and very low (line 12). Numerical values are used to define thresholds with comparators (line 13), whereas ordinal values are used to define values compared to all the services of an SBS under analysis (line 12). We define ordinal values with the box-plot statistical technique [39] to relate ordinal values with concrete metric values while avoiding setting artificial thresholds.

The metric suite (lines 9-11) encompasses both static and dynamic metrics. The static metric suite includes (but is not limited to) the following metrics: number of methods declared (`NMD`), number of incoming references (`NIR`), number of outgoing references (`NOR`), coupling (`CPL`), cohesion (`COH`), average number of parameters in methods (`ANP`), average number of primitive type parameters (`ANPT`), average num-

ber of accessor methods (ANAM), and average number of identical methods (ANIM). The dynamic metric suite contains: number of method invocations (NMI), number of transitive methods invoked (NTMI), response time (RT), and availability (A).

The DSL shown in Figure 2 can be extended by adding new metrics and–or values. Later in Section 4.6 (see Table 6), we specify and detect three new SOA antipatterns, which require four new metrics. We also add these metrics to our DSL. The new metrics (line 11) include: number of services encapsulated (NSE), total number of parameters (TNP), number of interfaces (NI), and number of utility methods (NUM). Utility methods provide facilities with logging, data validation, and–or notifications, rather than any business functionalities.

3.1.1. *Example of Rule Cards*

Figure 3 illustrates the grammar with the rule cards of the *Multi Service* and *Tiny Service* antipatterns. The *Multi Service* antipattern is characterized by very high response time and number of methods and low availability and cohesion. A *Tiny Service* corresponds to a service that declares a very low number of methods and has a high coupling with other services. For the sake of clarity, we illustrate the DSL with two intra-service antipatterns, *i.e.*, antipatterns within a service. However, the DSL allows also the specification of inter-service antipatterns, *i.e.*, antipatterns spreading over more than one service. We provide the rule cards of such other more complex antipatterns later in the experiments (see Section 4).

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

(a) Multi Service

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

(b) Tiny Service

Figure 3. Rule Cards for Multi Service and Tiny Service.

Using a DSL offers greater flexibility than implementing ad hoc detection algorithms, because it allows describing antipatterns using high-level, domain-related abstractions and focusing on *what* to detect instead of *how* to detect it [40]. Indeed, the DSL is independent of any implementation concern, such as the computation of static and dynamic metrics and the multitude of SOA technologies underlying SBSs. Moreover, the DSL allows the adaptation of the antipattern specifications to

the context and characteristics of the analysed SBSs by adjusting the metrics and associated values.

3.2. *Generation of Detection Algorithms*

From the specifications of the SOA antipatterns described with the DSL, we automatically generate the detection algorithms for all antipatterns like in [4]. In this paper, we improved the process of generation of detection algorithms. In [4], we used Kermeta [41] to automate the algorithm generation. However, in this paper, we used Ecore [42] and Acceleo [43] which are more recent meta-modeling frameworks than Kermeta.

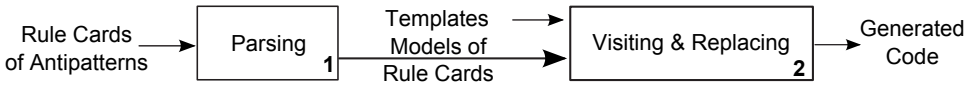

Figure 4. Generation of Detection Algorithms.

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 syntactically validate the rule card models against the meta-model of our DSL. Ecore also guarantees the correctness of the rule card models. We use a template-based code generation technique provided by Acceleo. To do this, we define a unique template for all rule cards consisting of well-defined tags to be replaced with the metric values defined in the rule cards of antipatterns. Finally, the template is applied to a rule card model resulting in a JAVA class, which is directly compilable and executable without any manual involvement.

Figure 5 shows the template for the *Multi Service* that we use to generate the detection code. In the first line of Figure 5, the template includes the meta-model of our DSL. It also provides *tags*, which are identified in square brackets and correspond to the variables to be replaced with rule card name, rule names, metrics, values, and different operators. Only one template is required for all the rule cards. Thus, it is easy to maintain them.

Figure 6 shows the detection code generated by Acceleo based on the rule card and the template defined in Figure 5. This generation creates a JAVA class with the operators and different metrics with their concrete values. The class generated here is directly compilable and executable using the JAVA classloader. Engineers only need to provide the concrete implementation of the metrics referred by the rule card.

This generative process is fully automated to avoid any manual tasks, which are usually repetitive and error-prone. This process also ensures the traceability between the specifications of antipatterns with the DSL and their concrete detection in SBSs using our underlying framework. Consequently, software engineers can focus

```
[module  generate('http://ruleCard/1.0')]

[template public generateElement(aRoot : Root)]
[for (aRuleCard : RuleCard | cards)]
[file (aRuleCard.name.concat('.java'), false)]
[comment @main /]
import com.sofa.metric.Metric;
import com.sofa.motifs.AMotif;
import com.sofa.rulecard.operators.Operator;

public class [aRuleCard.name/] extends AMotif {
    public [aRuleCard.name/](){
    [for (p : AbstractRule | rules)]
        [let m : Metric = p]
            [let relValue : RelativeValue = m.val]
        this.metrics.put(Metric.[m.id_metric/], "[relValue.value/]");
            [/let]
            [let expression : Expression = m.val]
        this.metrics.put(Metric.[m.id_metric/], "[expression.comparer/] [expression.threshold/]");
            [/let]
```

Figure 5. The Snapshot of the template for the *Multi Service*.

```
package com.soda.antipatterns;

import com.sofa.metric.Metric;
import com.sofa.motifs.AMotif;
import com.sofa.rulecard.setoperators.Operator;

public class MultiService extends AMotif {

    public MultiService() {
        this.operator = Operator.INTER;
        this.metrics.put(Metric.NMD, "VERY_HIGH");
        this.metrics.put(Metric.COH, "LOW");
        this.metrics.put(Metric.RT, "VERY_HIGH");
    }

}
```

Figure 6. The Snapshot of the generated code for the *Multi Service*.

on the specification of antipatterns, without considering any technical aspects of the underlying framework.

3.3. *SOFA: Underlying Framework*

We develop a framework, called SOFA (Service Oriented Framework for Antipatterns), that supports the detection of SOA antipatterns in SBSs. This framework, designed itself as an SBS and illustrated in Figure 7, provides different services corresponding to the main steps for the detection of SOA antipatterns: (1) the automated generation of detection algorithms; (2) the computation of static and dynamic metrics; and (3) the specification of rules including different sub-services

for the rule language, the box-plot statistical technique, and the set operators. The

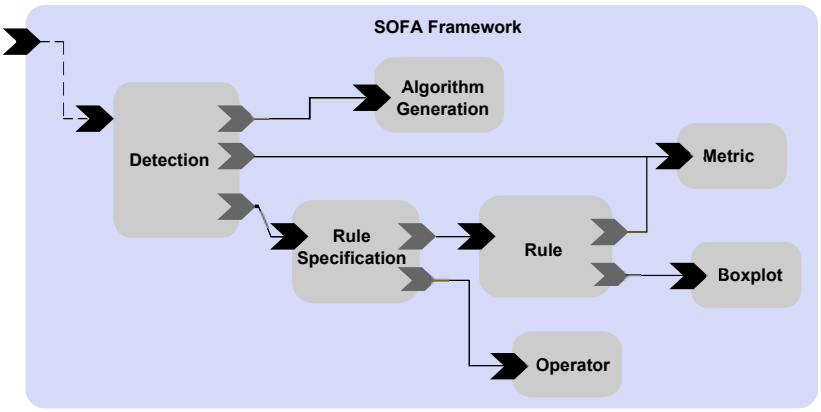

Figure 7. The SOFA Framework.

rule-specification and algorithm-generation services provide all constituents to describe models of rule cards as well as the algorithms to visit rule card models and to generate detection algorithms from these models. These different constituents rely on Model Driven Engineering techniques, which provide the means to define a DSL, parse it, and check its conformance with the grammar. We also use Ecore and Acceleo for generating the detection algorithms based on models of rule cards.

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

These sensors and triggers are provided at the deployment of the SBS under analysis. The code excerpt shown in Figure 8 presents the computation of the response time as a join point at the service interface level. The sensor `RTIntentHandler` (line 1) corresponds to an aspect that will intercept a service call and monitor the service response time. An intent join point (line 2) corresponds to the interface where a service invocation has been intercepted. The code enabling the computation of the response time is inserted before and after the invocation of the service (line 5). This new monitoring aspect is then declared as a service and added to the SOFA framework within the metric module (line 7).

SCA is a relatively new standard, advocated by researchers and major software vendors, like IBM and Oracle, for developing technology-agnostic and extensible SBSs. FraSCAti encompasses different modules for binding, deploying, running, and monitoring SBSs. Such a platform, *i.e.*, FraSCAti is essential for allowing the

```
1: public class RTIntentHandler implements IntentHandler {

    2: public Object invoke(IntentJoinPoint ijp) {

    3: long startTime = System.currentTimeMillis();
    4: Object ret = null;
    5: ret = ijp.proceed();
    6: long estimatedTime = System.currentTimeMillis() - startTime;

    7: Metrics.setValue("RT",estimatedTime);

    8: return ret;

  9: }

10: }
```

Figure 8. The Example of the usage of Sensors and Triggers.

detection of SOA antipatterns at execution time [6], which provides also runtime support for the SCA standard. Furthermore, the SOFA framework is implemented itself as an SCA component to ease its use and evolution and to offer it as a service to end-users concerned by the design and QoS of their SBSs. Currently, a prototype version of SOFA is available on http://sofa.uqam.ca/soda.html.

4. Experiments

To demonstrate the completeness and extensibility of our DSL, the accuracy of the generated algorithms, and the usefulness of the detection results with their related performance, we performed experiments with 10 antipatterns on two different service-based SCA systems, *i.e.*, *Home-Automation* and *FraSCAti* [5]. In our previous work [4], we presented detection results on *Home-Automation* only. In this paper, we perform similar experiments on *FraSCAti* while keeping the same assumptions as in [4].

4.1. *Assumptions*

The experiments aim at validating the following four assumptions:

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

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

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

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

4.2. *Subjects*

We apply our SODA approach using the SOFA framework to specify 10 different SOA antipatterns. Table 1 summarizes these antipatterns that we specified in [4], of which the first seven are from the literature and three others has been defined in our previous work, namely, *Bottleneck Service*, *Service Chain*, and *Data Service*. These new antipatterns are inspired from OO code smells [27]. In these summaries, we highlight in bold the key concepts relevant for the specification of their rule cards given in Figure 9.

4.3. *Objects*

We perform the experiments on two different service-based SCA systems, *i.e.*, *Home-Automation* and *FraSCAti* [5, 6]. *Home-Automation* has been developed independently for controlling remotely many basic household functions for elderly home-care support. It includes 13 services with a set of 7 predefined use-case scenarios for executing it at runtime. We use two versions of *Home-Automation*: the original version of the system, which includes 13 services, and a version modified by adding and modifying services to inject intentionally some antipatterns. The modifications on *Home-Automation* have been performed by an independent engineer to avoid biasing the results.

Figures 11 and 12 show the high-level design for *Home-Automation*[a] and *FraSCAti*[b] respectively. Among more than 100 services, Figure 12 shows only the principal service components of *FraSCAti*.

FraSCAti is itself implemented as an SCA system composed of 13 SCA composites, for a total of 91 SCA components, with 130 provided services. *FraSCAti* is the largest SCA system currently freely available. It is an open source implementation of the SCA standard, and supports components implementation with different technologies (BPEL, Java, Spring, OSGi, and Java supported languages, etc.) and also multiple binding technologies for communication between components (*i.e.*, RMI, SOAP, REST, JSON, JNA, and UPnP). Like *Home-Automation*, in *FraSCAti* we define a set of 6 scenarios and execute them to perform the detection.

[a] websvn.ow2.org/listing.php?repname=frascati&path=/trunk/demo/home-automation/
[b] frascati.ow2.org/doc/1.4/ch12s04.html

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

(a) Data Service

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

(b) The Knot

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

(c) Chatty Service

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

(d) Nobody Home

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

(e) Bottleneck Service

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

(f) Sand Pile

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

(g) Service Chain

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

(h) Duplicated Service

Figure 9. Rule Cards for Different Antipatterns.

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

Multi Service also known as *God Object* corresponds to a service that implements a ***multitude of methods*** related to different business and technical abstractions. This 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 of it is ***overloaded***, which may also induce a ***high response time*** [18].
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 [18].
Sand Pile is also known as *"Fine-Grained Services"*. It appears when a service is ***composed*** by multiple smaller services sharing ***common data***. It thus has a ***high data cohesion***. The common data shared may be located in a ***Data Service*** antipattern (see below) [3].
Chatty Service corresponds to a set of services that exchange a lot of ***small data of primitive types***, usually with a ***Data Service*** antipattern. The *Chatty Service* is also characterized by a ***high number of method invocations***. *Chatty Services* chat a lot with each other [18].
The Knot is a set of ***very low cohesive*** services, which are ***tightly coupled***. These services are thus less reusable. Due to this complex architecture, the ***availability*** of these services may be ***low***, and their ***response time high*** [9].
Nobody Home corresponds to a service, defined but actually never used by clients. Thus, the ***methods*** from this service are ***never invoked***, even though it may be ***coupled*** to other services. Yet, it still requires deployment and management, despite of its non-usage [16].
Duplicated Service, a.k.a. *The Silo Approach*, introduced by IBM, corresponds to a set of ***highly similar*** services. Because services are implemented multiple times as a result of the silo approach, there may have ***common*** or ***identical methods*** with the ***same names and–or parameters*** [15].
Bottleneck Service is a service that is highly used by other services or clients. It has a ***high incoming and outgoing coupling***. Its ***response time*** can be ***high*** because it may be used by too many external clients, for which clients may need to wait to get access to the service. Moreover, its ***availability*** may also be low due to the traffic.
Service Chain a.k.a. *Message Chain* [27] in OO systems corresponds to a ***chain of services***. The *Service Chain* appears when clients request consecutive service invocations to fulfill their goals. This kind of ***dependency*** chain reflects the subsequent ***invocation*** of services.
Data Service, a.k.a. *Data Class* [27] in OO systems, corresponds to a service that contains ***mainly accessor methods***, *i.e.*, getters and setters. In the distributed applications, there can be some services that may only perform some simple information retrieval or ***data access*** to such services. *Data Services* contain usually ***accessor methods*** with ***small parameters*** of ***primitive types***. Such service has a ***high*** data ***cohesion***.

Table 3 shows different structural properties for the two different versions of *Home-Automation* and *FraSCAti*. Details on the two versions of *Home-Automation* system including all the scenarios and involved services are available online at http://sofa.uqam.ca/soda. More details on *FraSCAti* design are also available on the FraSCAti Web site (http://frascati.ow2.org).

Table 2. List of Three New Antipatterns.

God Component in SCA technology corresponds to a component that ***encapsulates*** a ***multitude of services***. This component represents high responsibility enclosed by ***many methods*** with ***many*** different types of ***parameters*** to exchange. It may have a ***high coupling*** with the communicating services. Being at the component-level, *God Component* is at a higher level of abstraction than the *Multi Service*, which is at the service-level, and usually ***aggregates*** a ***set of services*** [18].
Bloated Service is an antipattern related to service implementation where services in SOA become 'blobs' with ***one large interface*** and ***lots of parameters***. *Bloated Service* performs ***heterogeneous operations*** with ***low cohesion*** among them. It results in a system with less maintainability, testability, and reusability within other business processes. It requires the consumers to be aware of many details (*i.e.*, parameters) to invoke or customize them [17].
Stovepipe Service is an antipattern with ***large number*** of ***private*** or ***protected*** methods that primarily focus on performing infrastructure and ***utility functions*** (*i.e.*, logging, data validation, notifications, etc.) and few business processes (*i.e.*, data type conversion), rather than focusing on main operational goals (*i.e.*, very ***few public methods***). This may result in services with ***duplicated code***, longer development time, inconsistent functioning, and poor extensibility [18].

```
1 RULE_CARD: GodComponent {
2  RULE: GodComponent {INTER HighEncapsulatedService MultiMethod HighParameter};
3  RULE: HighEncapsulatedService {NOSE HIGH};
4  RULE: MultiMethod {NMD VERY_HIGH};
5  RULE: HighParameter {TNP VERY_HIGH};
6 };
```

(a) God Component

```
1 RULE_CARD: BloatedService {
2  RULE: BloatedService {INTER SingleInterface MultiMethod HighParameter LowCohesion};
3  RULE: SingleInterface {NOI EQUAL 1};
4  RULE: MultiMethod {NMD VERY_HIGH};
5  RULE: HighParameter {TNP VERY_HIGH};
6  RULE: LowCohesion {COH LOW};
7 };
```

(b) Bloated Service

```
1 RULE_CARD: StovepipeService {
2  RULE: StovepipeService {INTER HighUtilMethod FewMethod DuplicatedCode};
3  RULE: HighUtilMethod {NUM VERY_HIGH};
4  RULE: FewMethod {NMD VERY_LOW};
5  RULE: DuplicatedCode {ANIM HIGH};
6 };
```

(c) Stovepipe Service

Figure 10. Rule cards for three new Antipatterns

4.4. *Process*

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

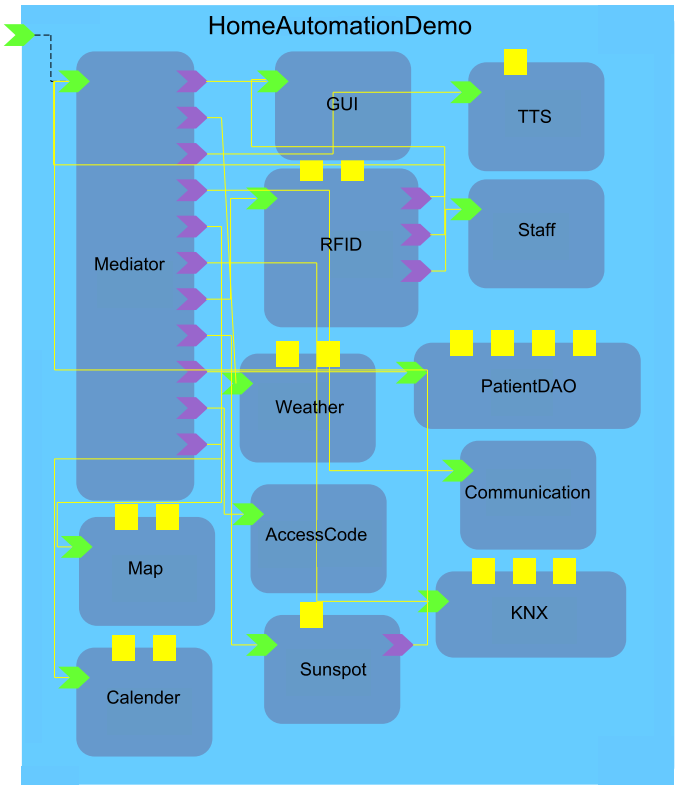

Figure 11. The Design of Home-Automation System.

Table 3. Structural properties for the two versions of *Home-Automation* and *FraSCAti* (NOS: Number of Services, NOM: Number of Methods, NOC: Number of Classes).

System Name	Version	Size	NOS	NOM	NOC
Home Automation 1.0	original	3.2KLOC	13	226	48
Home Automation 1.1	evolved	3.4KLOC	16	243	52
FraSCAti	original	26.75KLOC	130	1882	403

on *FraSCAti*. Finally, we validated the detection results by analysing the suspicious services manually to: (1) validate that these suspicious services are true positives and (2) identify false negatives (if any), *i.e.*, missing antipatterns. For this last validation step, we use the measures of precision (Equation 1), recall (Equation 2) and F_1-measure (Equation 3) [44]. Precision estimates the ratio of true antipatterns identified among the detected antipatterns, while recall estimates the ratio of detected antipatterns among the existing antipatterns.

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

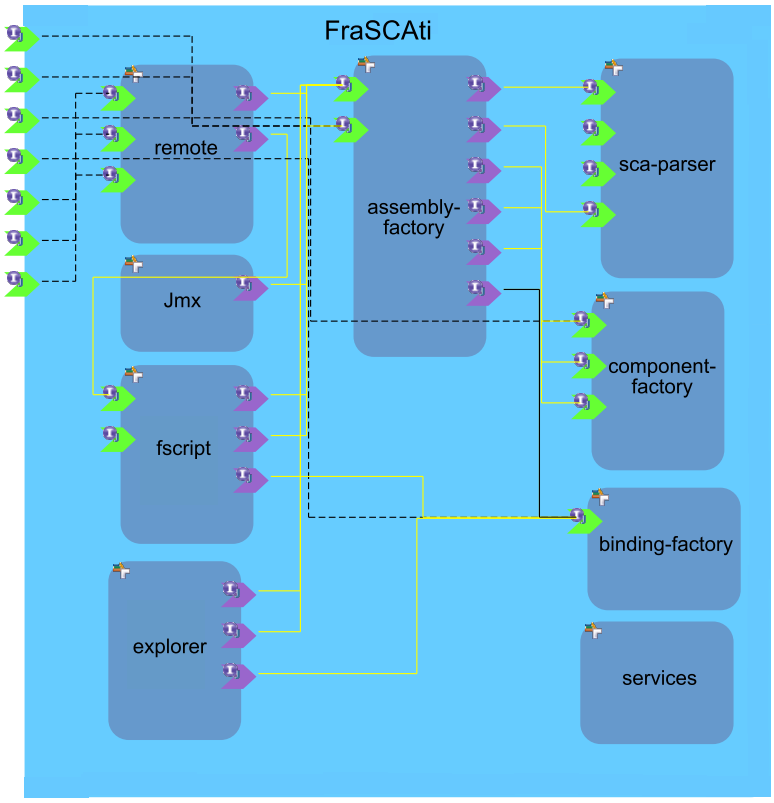

Figure 12. The Design of FraSCAti taken from FraSCAti Web site (each component represents an SCA composite here).

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

We also compute F_1-measure, *i.e.*, the weighted average of precision and recall, to measure the accuracy of our detection algorithms.

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

The validation for *Home-Automation* and *FraSCAti* has been carried out manually by an independent software engineer. We provided him the complete descriptions of the target SOA antipatterns, and the complete design of the two versions of *Home-Automation* and *FraSCAti*. Manual validation is laborious task and may require 1 to 3 days for an engineer to validate depending on the size of the target SBS. We also reported the *FraSCAti* detection results to its development team. The *FraSCAti* development team provided their feedback stating if they agree or not with our preliminary detection results including the *Multi Service*, *Tiny Service*,

Chatty Service, *The Knot*, *Bottleneck Service*, and *Data Service*. They validate our detection results based on the antipatterns textual descriptions and the rule cards we defined, and they agreed with us on five out of these six antipatterns. They do not validate the *Multi Service* because of the subjectivity of the rule card, that we modified afterwards.

4.5. *Results*

We present here the detection results for *Home-Automation* and *FraSCAti* followed by a general discussion. We then discuss the assumptions for both the systems in Section 4.6.

4.5.1. *Results on Home-Automation*

Table 4 presents the results for the detection of the 10 SOA antipatterns on the original and evolved version of *Home-Automation* that we also presented in [4]. For each antipattern, the table reports the involved services in the second column, the analysis method: *static* (S) and–or *dynamic* (D) in the third, then the metric values of rule cards in the fourth, and finally the detection times in the fifth. The three last columns report the precision, recall, and F_1-measure.

4.5.2. *Details of the Results on Home-Automation*

We briefly present the detection results of the *Tiny Service* and *Multi Service*. The service `IMediator` has been identified as a *Multi Service* because of its very high number of methods (*i.e.*, `NMD` equal 13) and its low cohesion (*i.e.*, `COH` equal 0.027). These metric values have been evaluated by the *Boxplot* service respectively as high and low in comparison with the metric values of other services of *Home-Automation*. For example, for the metric `NMD`, the *Boxplot* estimates the median value of `NMD` in *Home-Automation* as equal to 2. In the same way, the detected *Tiny Service* has a very low number of methods (*i.e.*, `NMD` equal 1) and a high coupling (*i.e.*, `CPL` equal 0.44) with respect to other values. The values of the cohesion `COH` and coupling `CPL` metrics range from 0 to 1. In the original version of *Home-Automation*, we did not detect any *Tiny Service*. We then extracted one method from `IMediator` and moved it in a new service named `MediatorDelegate` and then this service has been detected as a *Tiny Service*.

We also detected seven other antipatterns within the original version of *Home-Automation* (see Table 4), namely, *Duplicated Service*, *Chatty Service*, *Sand Pile*, *The Knot*, *Bottleneck Service*, *Data Service*, and *Service Chain*. All these antipatterns involve more than one service, except *Data Service* and *Duplicated Service*. The service `PatientDAO` has been detected as a *Data Service* because it performs simple data accesses. Moreover, in the evolved version, we detected the *Nobody Home* antipattern, after an independent developer introduced the service `UselessService`, which is defined but never used in any scenarios. We detected a

Table 4. Results for the detection of 10 SOA Antipatterns in the original and evolved version of *Home-Automation* system *(S: Static, D: Dynamic).*

AntipatternName	ServicesInvolved	Analysis	Metrics	Time	Precision	Recall	F_1

Tiny Service	`MediatorDelegate`	`S`	`NOR=4`
			`CPL=0.440`
			`NMD=1`
				0.194s	[1/1]
					100%
						[1/1]
						100%
							100%

Multi Service	`IMediator`	`S, D`	`COH=0.027`
			`NMD=13`
			`RT=132ms`
				0.462s	[1/1]
					100%
						[1/1]
						100%
							100%

Duplicated Service	`CommunicationService` `IMediator`	`S`	`ANIM=25%`	0.215s	[2/2] 100%	[2/2] 100%	100%

Chatty Service	`IMediator`
	`PatientDAO`
		`S, D`	`ANP=1.0;ANPT=1.0;`
			`NMI=3;ANAM=100%;`
			`COH=0.167`
			`ANP=1.0;ANPT=1.0;`
			`NMI=3;ANAM=100%;`
			`COH=0.167`
				0.383s	[2/2]
					100%
						[2/2]
						100%
							100%

Nobody Home	`UselessService`	`S, D`	`NIR>0` `NMI=0`	1.154s	[1/1] 100%	[1/1] 100%	100%

Sand Pile	`HomeAutomation`	`S`	`NCS=13`
			`ANP=1.0`
			`ANPT=1.0`
			`ANAM=100%`
			`COH=0.167`
				0.310s	[1/1]
					100%
						[1/1]
						100%
							100%

The Knot	`IMediator`
	`PatientDAO`
		`S, D`	`COH=0.027`
			`NIR=7`
			`NOR=7`
			`CPL=1.0`
			`RT=57ms`
				0.412s	[1/2]
					50%
						[1/1]
						100%
							66.67%

Bottleneck Service	`IMediator`
	`PatientDAO`
		`S, D`	`NIR=7;NOR=7;`
			`CPL=1.0;RT=40ms`
			`NIR=4;NOR=4;`
			`CPL=0.57;RT=2ms`
				0.246s	[2/2]
					100%
						[2/2]
						100%
							100%

Data Service	`PatientDAO`	`S`	`ANAM=100%`
			`COH=0.167`
			`ANPT=1.0`
			`ANP=1.0`
				0.268s	[1/1]
					100%
						[1/1]
						100%
							100%

Service Chain	`IMediator`
	`SunSpotService`
	`PatientDAO`
	`PatientDAO2`
		`D`	`NTMI=4`	0.229s	[3/4]
					75%
						[3/3]
						100%
							85.71%

AVERAGE				0.387s	[15/17] 92.5%	[15/15] 100%	95.24%

consecutive chain of invocations of `IMediator` → `SunSpotService` → `PatientDAO` → `PatientDAO2`, which forms a *Service Chain*, whereas engineers validated `IMediator` → `PatientDAO` → `PatientDAO2`. Therefore, we had precision of 75% and recall of 100% for the *Service Chain* antipattern. Therefore, we had precision of 75% and recall of 100% for the *Service Chain* antipattern. The `SunSpotService` was not validated by the engineers and thus, it was considered as a false positive. However, the detected chain exists in the system but only in one scenario. Engineers did not consider the full chain as harmful and therefore did not classify it as an antipattern. Also, our detection algorithm reported `PatientDAO` service as the *Knot* antipattern because the availability (`A`) value was discarded, which was very high, *i.e.*, 100%. In contrast, we specified the availability (`A`) in the *Knot* antipattern as *low*. Moreover,

we detected the `HomeAutomation` itself as *Sand Pile*.

4.5.3. *Results on FraSCAti*

We perform another experiment with *FraSCAti* as a new contribution with respect to *Home-Automation* from [4]. Table 5 presents the results for the detection of the 10 SOA antipatterns in *FraSCAti*. For each antipattern, the table also reports the suspicious services in the second column, the metric values of rule cards in the third, and finally the detection times in the fourth. The three last columns report the precision, recall, and F_1-measure.

Table 5. Results for the detection of 10 SOA Antipatterns in the original version of *FraSCAti (S: Static, D: Dynamic)*.

AntipatternName	ServicesInvolved	Analysis	Metrics	Time	Precision	Recall	F_1
Tiny Service	`sca-parser`	`S`	`NMD=1;CPL=0.56`	0.067s	[1/1] 100%	[1/1] 100%	100%
Multi Service	`juliac` `Explorer-GUI`	`S, D`	`COH=0.1;NMD=5;RT=1018ms` `n/a`	0.066s	[1/1] 100%	[1/2] 50%	66.67%
Duplicated Service	*not present*	`S`	`n/a`	0.302s	[0/0] 100%	[0/0] 100%	100%
Chatty Service	*not present*	`S, D`	`n/a`	0.057s	[0/0] 100%	[0/0] 100%	100%

Nobody Home	`NativeCompiler`
	`ServletManager`
	`WsdlCompiler`
	`BPELEngine`
		`S, D`	`NMI=0;NIR>0`
			`NMI=0;NIR>0`
			`NMI=0;NIR>0`
			`NMI=0;NIR>0`
				0.057s	[3/4] 75%	[3/3] 100%	85.71%

Sand Pile	*not present*	`S`	`n/a`	0.058s	[0/0] 100%	[0/0] 100%	100%
The Knot	`sca-parser`	`S, D`	`CPL=0.84;COH=0.08;RT=44ms`	0.07s	[1/1] 100%	[1/1] 100%	100%

Bottleneck Service	`sca-composite`
	`sca-parser`
		`S, D`	`RT=41ms;CPL=0.96;`
			`NIR=16;NOR=8`
			`RT=45ms;CPL=0.84;`
			`NIR=16;NOR=5`
				0.086s	[1/2] 50%	[1/1] 100%	66.67%

Data Service	*not present*	`S`	`n/a`	0.07s	[0/0] 100%	[0/0] 100%	100%

Service Chain	`MembraneGeneration`
	`Processor`
	`ComponentFactory`
	`MembraneGeneration`
	`TypeFactory`
	`Processor`
	`ComponentFactory`
	`MembraneGeneration`
	`Processor`
	`Processor`
	`BindingFactory`
	`PluginResolver`
		`D`	`NTMI=4`
			`NTMI=4`
			`NTMI=4`
				0.056s	[2/3] 66.67%	[2/2] 100%	80%

Average				0.089s	89.17%	95%	89.91%

4.5.4. *Details of the Results on FraSCAti*

We discuss some interesting detection results of *FraSCAti* reported in Table 5. We detect `juliac` as *Multi Service* because of its very high response time (`RT` equal 1,018ms), low cohesion (`COH` equal 0.1), and high number of methods declared (`NMD` equal 5). The median values estimated by the *Boxplot* service, *i.e.*, median of `RT` is 4ms, `COH` is 0.1, and `NMD` is 1 compared to other services in *FraSCAti* classify `juliac` as *Multi Service*. `juliac` is likely a true positive because it implements six different features belonging to two different abstractions. Indeed, `juliac` provides services for the membrane generation and the Java compilers. Therefore, it is low cohesive and highly used because each component needs a membrane, and each composite file needs to be compiled by the Java compiler. Moreover, these actions are resource-consuming and requires more execution time than other services. Manual inspection by the engineer also validated this detection.

Subsequently, the inspection of *FraSCAti* also allowed the identification of `Explorer-GUI` as a *Multi Service*. The *FraSCAti* development team confirmed that this service uses a high number of other services provided by *FraSCAti*. Indeed, this component encapsulates the graphical interface of `FraSCAti Explorer`, which aims to provide an exhaustive interface of FraSCAti functionalities. SOFA was not able to detect it because the execution scenarios did not involve the graphical interface of `FraSCAti Explorer`. Therefore, with one service detected as true positive, and with one missing occurrence of *Multi Service*, *i.e.*, `Explorer-GUI`, we have a precision of 100% and recall of 50%.

We also detect `sca-parser` as the *Tiny Service* with its small number of methods (`NMD` equal 1) and a high coupling (`CPL` equal 0.56). The engineer and the FraSCAti team also validated this detection. The boxplot median values are respectively 1 and 0.11 for `NMD` and `CPL`. After the manual inspection of *FraSCAti* implementation, the independent engineer identified that `sca-parser` contains only one method, *i.e.*, `parse(QName` *`qname`*`, ParsingContext` *`parsingContex`*`t)`. In some cases, for a given metric, the median value and the high and–or low value might be identical if the values of most services are equal. For example, the median and low values are the same for `NMD` because out of 86 analysed services, 50 services have the `NMD` value of 1. While concerned about the coupling CPL, `sca-parser` has dependency references to five other services (*i.e.*, `sca-metamodel-*`) that give a high coupling. In fact, the coupling CPL values presented here are calculated on the logarithmic scale, *i.e.*, the more references a service has to other services, the more highly coupled it is. The FraSCAti development team also validated `sca-parser` as a *Tiny Service*. However, according to them, `sca-parser` is invoked alone when only a reading for an SCA composite file is requested. However, *FraSCAti* performs more tasks than just reading and–or parsing an SCA composite file, and these other tasks are performed by other services such as `AssemblyFactory`. These several delegation also explains the high outgoing coupling. The appropriate detection of only one service as a *Tiny Service* leads to a precision and recall of 100%.

The `sca-composite` and `sca-parser` services are detected as *Bottleneck Service* in *FraSCAti* with a high response time (*i.e.*, `RT` values are 41ms and 45ms, respectively) and a very high coupling (*i.e.*, `CPL` values are 0.96 and 0.84). The coupling `CPL` is calculated by means of `NIR` and `NOR`, which are also very high for these two services. As estimated by the *Boxplot* service, the median for `CPL` and `RT` are 0.68 and 5ms respectively. After manually analysing the FraSCAti design, the engineer found that only the `sca-parser` service is highly used by other services, and validated `sca-parser` as the only *Bottleneck Service*. The *FraSCAti* development team also agreed with this manual detection result, and confirmed that if too many external clients try to invoke the `sca-parser` service, they might wait to get the access. The precision and recall for *Bottleneck Service* are 50% and 100% respectively, with one false positive. The `sca-composite` is detected as *Bottleneck Service* and is a false positive again because of the value of availability (`A`), which is always very high, *i.e.*, 100%. For an SBS with several Composites, while the *FraSCAti* invokes the `sca-parser` service to parse all the relevant Composites, the `sca-parser` service naturally has the low availability. This low availability is mainly due to multiple invocations simultaneously by several clients to parse their Composites using the same `sca-parser`. Since, we have the availability (`A`) of 100% for all the components and did not consider the value of availability while reporting suspicious services, the `sca-parser` was reported as *Bottleneck Service*.

We also report three other detected SOA antipatterns in *FraSCAti*, namely {`NativeCompiler, ServletManager, WsdlCompiler,` and `BPELEngine`} as *Nobody Home*, `sca-parser` as *The Knot*, and {`MembraneGeneration, TypeFactory,` and `Processor`} as involved in a *Service Chain*. The `BPELEngine` has been detected as a *Nobody Home* antipattern because its implementation does not support the weaving of non-functional code such as sensors and triggers (see section 3.3). In fact, all the BPEL implementations of *FraSCAti* are, by default, discarded from the weaving algorithms provided by *FraSCAti*. We also detected one false positive for the *Service Chain* (`Processor` → `Processor` → `BindingFactory` → `PluginResolver`) that was not confirmed by the engineers. This detection was due to the fact that the `Processor` service calls itself using a public method and artificially extends the chain of calls. We may consider the modification of the *Service Chain* detection algorithm in order to eliminate self calls. Our detection algorithms do not detect any *Duplicated Service*, *Chatty Service*, *Sand Pile*, and *Data Service* antipatterns in *FraSCAti*. The absence of *Sand Pile* and *Chatty Service* is obvious, as they are related to *Data Service* and evidently, there is no such antipattern in *FraSCAti*, *i.e.*, as confirmed by the *FraSCAti* team and our detection results. Our detection algorithms do not identify any suspicious services, after calculating the metric values we define in rule cards for those antipatterns. Indeed, as validated by FraSCAti team, there are no *Duplicated Service* and *Data Service* antipatterns in *FraSCAti*, and our detection algorithms do not filter any services as false positives either. Therefore, we obtain a precision and recall of 100%. However, like in *Home-Automation*, the availability (`A`) for the services in *FraSCAti* is 100%, because services are deployed

locally.

In summary, very few services, *i.e.*, 10 are actually involved in 6 antipatterns (4 antipatterns are not present) in *FraSCAti*, in comparison to the high number of services, *i.e.*, 130 services. *FraSCAti* is well designed with continuous maintenance and evolution. Mostly, services (*e.g.,* `sca-parser` and `sca-composite-*`) related to parsing and handling the composite file are involved in the antipatterns. The presence of such antipatterns in a system is not surprising because there is no other way to develop a parser without introducing a high coupling among services. More detailed on *FraSCAti* results are available at http://sofa.uqam.ca/soda.html.

4.6. *Discussion on the Assumptions*

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

A1. Generality: *The DSL allows the specification of many different SOA antipatterns, from simple to more complex ones.* Using our DSL, we specified 10 SOA antipatterns described in Table 1, as shown in rule cards given in Figure 3 and 9. These antipatterns range from simple ones, such as the *Tiny Service* and *Multi Service*, to more complex ones such as the *Bottleneck* and *Sand Pile*, which involve several services and complex relationships. In particular, *Sand Pile* has both the `ASSOC` and `COMPOS` relation types. Also, both *Sand Pile* and *Chatty Service* refer in their specifications to another antipattern, namely *DataService*. Thus, we show that we can specify from simple to complex antipatterns, which support the generality of our DSL.

A2. Accuracy: *The generated detection algorithms have a recall of 100%, i.e., all existing antipatterns are detected, and a precision greater than 75%, i.e., more than three-quarters of detected antipatterns are true positive.* For *Home-Automation*, as indicated in Table 4, we obtain a recall of 100%, which means all existing antipatterns are detected, whereas the precision is 92.5%. We have high precision and recall because the analysed system, *Home-Automation* is a small SBS with 13 services. Also, the evolved version includes two new services. Therefore, considering the small *but* significant number of services and the well defined rule cards using DSL, we obtain such a high precision and recall. For the original *Home-Automation* version, out of 13 services, we detected 6 services that are responsible for 8 antipatterns. Besides, we detected 2 services (out of 15) that are responsible for 2 other antipatterns in the evolved system.

Furthermore, for *FraSCAti*, as shown in Table 5, we achieve a recall of 95% and a precision of 89.17%. The accuracy of our detection algorithms is acceptable, considering the high of number of services in *FraSCAti*, *i.e.*, 130 services (see Table 3). According to our detection results, in total 15 services are responsible for 10 antipatterns in *FraSCAti*. We achieve our expected accuracy for detecting antipatterns even for a large SBS like *FraSCAti* and can support the second assumption.

Table 6. List of new metrics added to Our DSL

Metric ID	Description
NSE	Number of Services Encapsulated
TNP	Total Number of Parameters
NI	Number of Interfaces
NUM	Number of Utility Methods

A3. Extensibility: *The DSL and the SOFA framework are extensible for adding new SOA antipatterns.* The DSL has been initially designed for specifying the seven antipatterns described in the literature (see Table 1). Then, through inspection of the SBS and inspiration from OO smells, we added three new antipatterns, namely the *Bottleneck Service*, *Service Chain*, and *Data Service*. When specifying these new antipatterns, we reused four already-defined metrics and we added in the DSL and SOFA four more metrics (`ANAM`, `NTMI`, `ANP`, and `ANPT`). The language is flexible in the integration of new metrics. However, the underlying SOFA framework should also be extended to provide the operational implementations of the new metrics.

To further show the extensibility of our DSL and our framework, apart from the antipatterns in [4], we also specify three more new SOA antipatterns in this paper, namely *God Component*, *Bloated Service*, and *Stovepipe Service*. Table 2 summarizes these three newly specified antipatterns and Figure 10 provides the corresponding rule cards. We defined four new metrics for specifying these new antipatterns. Table 6 lists these new metrics. The results of their detection on *FraSCAti* is reported in Table 7. We report the suspicious services in column 2, and corresponding metric values for each new antipattern in column 4. We also compute and report the precision, recall and F_1-measure for each antipattern (columns 6-8). Such an addition can only be realized by skilled developers with our framework, which may require from 1 hour to 2 days according to the complexity of the metrics.

Thus, by extending the DSL with these three new antipatterns and integrating them within the SOFA framework for the detection, we support A3.

Table 7. Results for the detection of 3 new SOA Antipatterns in the original version of *FraSCAti (S: Static)*

Antipattern	ServicesInvolved	Analysis	Metrics	Time	Precision	Recall	F_1
God Component	`FraSCAti` `component-factory`	*S*	`NOSE=6;NMD=12;TNP=12` `NOSE=5;NMD=7;TNP=12`	0.069s	[2/2] 100%	[2/2] 100%	100%

Bloated Service	`component-factory`
	`factory`
	`frascati-binding-http`
		S	`NOI=1;NMD=7;`
			`TNP=12;COH=0.066`
			`NOI=1;NMD=7;`
			`TNP=12;COH=0.066`
			`NOI=1;NMD=5;`
			`TNP=8;COH=0.065`
				0.071s	[3/3]
					100%
						[3/3]
						100%
							100%

Stovepipe Service	*not present*	*S*	`n/a`	0.081s	[0/0] 100%	[0/0] 100%	100%

A4. Performance: *The computation time required for the detection of antipatterns*

using the generated algorithms is reasonably very low, i.e., in the order of few seconds. We perform all experiments on an Intel Dual Core at 3.30GHz with 4GB of RAM. Computation times include computing metric values, introspection delay during static and dynamic analyses, and applying detection algorithms, and do not include the time while targeted services are processing or the time belonging to availability or response time metrics. The computation times for the detection of antipatterns in *Home-Automation* is reasonably low, *i.e.*, ranging from 0.194s to 1.154s with an average of 0.387s (see Table 4).

Moreover, even for a large scale SBS like *FraSCAti*, we record a bit lower detection time, *i.e.*, ranging from 0.056s to 0.302s with an average of 0.089s. Given a larger SBS, the detection time for *FraSCAti* should be higher than *Home-Automation*. However, we improved the SOFA implementation since our earlier work [4]. Previously in [4], we compute every metric for each antipattern whenever they appear in the rule cards. In this paper, for *FraSCAti*, we improve the way SOFA compute different metrics. Indeed, we now compute all the metrics only once and use them for various rule cards. This new implementation strategy substantially reduces the detection time.

The detection times depend mostly on computing some static metrics, such as `NMD`, `CPL`, `COH`, or `ANIM`. Computing those metrics requires thorough inspection of source code and sometimes a high number of pairwise comparisons among method signatures and/or parameters. However, the complexity of our detection algorithms are *linear*, *i.e.*, $O(\mathrm{n})$, with n represents the number of rules. Therefore, the cumulative detection times will increase with the number of rules in each rule card, and with the number of antipatterns to be detected in the target system. Such low computation times suggest that SODA could be applied on SBSs even with a larger number of services. Thus, we show that we support the fourth assumption positively.

4.7. *Threats to Validity*

The main threat to the validity of our results in [4] concerned their *external validity*, *i.e.*, the possibility to generalise our approach to other SBSs. In this paper, we minimize the external validity by experimenting with another large scale SCA system, *i.e.*, *FraSCAti*. As a part of the future work, we plan to experiment with other SBSs, if available. For *internal validity*, the detection results depend on the services provided by the SOFA framework, and also on the antipattern specifications using rule cards. We performed experiments on a representative set of antipatterns to lessen this threat to the internal validity. The subjective nature of specifying and validating antipatterns is a threat to *construct validity*. We try to lessen this threat by defining rule cards based on thorough literature review and domain analysis, and by involving an independent engineer and the *FraSCAti* team in the validation. We minimize *reliability validity* by automating the generation of the detection algorithms, where each subsequent detection produce consistent sets of results with

anticipated precision and recall.

5. Conclusion and Future Work

The specification and detection of SOA antipatterns are important to assess the design and QoS of SBSs and thus ease the maintenance and evolution of SBSs. In this paper, we presented a novel approach, named SODA, for the specification and detection of SOA antipatterns, and SOFA, its underlying framework. We proposed a DSL for specifying SOA antipatterns and a process for automatically generating detection algorithms from the antipattern specifications. We applied and validated SODA with 10 different SOA antipatterns on two SCA systems developed independently: (i) an original and an evolved version of *Home-Automation* with 13 services and (ii) *FraSCAti*, an open-source implementation of the SCA standard containing more than 100 services (almost 10 times bigger than *Home-Automation*). We demonstrated the usefulness of our approach and discussed its precision and recall.

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

Acknowledgments.

The authors thank Yousri Kouki and Mahmoud Ben Hassine for their help with respectively an early prototype of SOFA and the implementation of *Home-Automation*. The authors are also thankful to the FraSCAti development team and Benjamin Joyen Conseil for validating the results. This work is partly supported by the NESSOS European Network of Excellence and a NSERC Discovery Grant. And, this work is in memory of Anne-Françoise Le Meur, our dearly departed colleague, who initiated the work.

Bibliography

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

[2] George Spanoudakis and Khaled Mahbub. Requirements Monitoring for Service-based Systems: Towards a Framework based on Event Calculus. In *Automated Soft-*

ware Engineering, 2004. Proceedings. 19th International Conference on, pages 379–384, 2004.

[3] Jaroslav Král and Michal Žemlička. Crucial Service-Oriented Antipatterns. volume 2, pages 160–171. International Academy, Research and Industry Association (IARIA), 2008.

[4] Naouel Moha, Francis Palma, Mathieu Nayrolles, Benjamin Joyen Conseil, Yann-Gaël Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel. Specification and Detection of SOA Antipatterns. *10th International Conference on Service Oriented Computing*, November 2012.

[5] Lionel Seinturier, Philippe Merle, Damien Fournier, Valerio Schiavoni, Christophe Demarey, Nicolas Dolet, and Nicolas Petitprez. FraSCAti - Open SCA Middleware Platform v1.4. http://frascati.ow2.org/.

[6] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Softw. Pract. Exper.*, 42(5):559–583, May 2012.

[7] Thomas Erl. *SOA Design Patterns*. Prentice Hall PTR, January 2009.

[8] Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, November 2011.

[9] Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan. *SOA Patterns*. Manning Publications Co., 2012. To be published in Summer 2012.

[10] Bipin Upadhyaya, Ran Tang, and Ying Zou. An Approach for Mining Service Composition Patterns from Execution Logs. *Journal of Software: Evolution and Process*, 2012.

[11] Qianhui Althea Liang, Jen-Yao Chung, Steven Miller, and Yang Ouyang. Service Pattern Discovery of Web Service Mining in Web Service Registry-Repository. In *Proceedings of the IEEE International Conference on e-Business Engineering*, ICEBE '06, pages 286–293, Washington, DC, USA, 2006. IEEE Computer Society.

[12] A. J. M. M. Weijters and W. M. P. van der Aalst. Rediscovering Workflow Models from Event-based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, April 2003.

[13] Remco Dijkman, Marlon Dumas, Luciano Garcı́a-Bañuelos, and Reina Kääärik. Aligning Business Process Models. In *IEEE International Enterprise Distributed Object Computing Conference*, pages 45–53, September 2009.

[14] Mirjam Minor, Alexander Tartakovski, and Ralph Bergmann. Representation and Structure-Based Similarity Assessment for Agile Workflows. In *Proceedings of the 7th international conference on Case-Based Reasoning: Case-Based Reasoning Research and Development*, ICCBR '07, pages 224–238, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] Luba Cherbakov, Mamdouh Ibrahim, and Jenny Ang. SOA Antipatterns: The Obstacles to the Adoption and Successful Realization of Service-Oriented Architecture. www.ibm.com/developerworks/webservices/library/ws-antipatterns/.

[16] Steve Jones. SOA Anti-patterns. www.infoq.com/articles/SOA-anti-patterns, June 2006.

[17] Tarak Modi. SOA Management: SOA Antipatterns. www.ebizq.net/topics/soa_management/features/7238.html, August 2006.

[18] Bill Dudney, Stephen Asbury, Joseph K. Krozak, and Kevin Wittkopf. *J2EE AntiPatterns*. John Wiley & Sons Inc, August 2003.

[19] Nikola Trčka, Wil M. Aalst, and Natalia Sidorova. Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In *Proceedings of the 21st International Con-*

ference on Advanced Information Systems Engineering, CAMISE '09, pages 425–439, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transaction on Software Engineering*, 36(1):20–36, January 2010.

[21] Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni. Design Defects Detection and Correction by Example. In *IEEE 19th International Conference on Program Comprehension (ICPC)*, pages 81 –90, june 2011.

[22] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. Deviance From Perfection is a Better Criterion Than Closeness To Evil When Identifying Risky Code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE'10, pages 113–122, New York, NY, USA, 2010. ACM.

[23] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[24] Matthew James Munro. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[25] Dimitrios L. Settas, Georgios Meditskos, Ioannis G. Stamelos, and Nick Bassiliades. SPARSE: A Symptom-based Antipattern Retrieval Knowledge-based System using Semantic Web Technologies. *Expert Systems with Applications*, 38(6):7633–7646, June 2011.

[26] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.

[27] Martin J. Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[28] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

[29] Sunny Wong, Melissa Aaron, Jeffrey Segall, Kevin Lynch, and Spiros Mancoridis. Reverse Engineering Utility Functions Using Genetic Programming to Detect Anomalous Behavior in Software. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 141–149, Washington, DC, USA, 2010. IEEE Computer Society.

[30] Trevor Parsons and John Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*, 7(3):55–90, April 2008.

[31] cbsdetector. Technical report, July 2010.

[32] Bill Pugh, Andrey Loskutov, and Keith Lea. FindBugs. Technical report, December 2012.

[33] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An Integrated Platform for Quality Assessment of Object-oriented Design. pages 77–80. Society Press, 2005.

[34] JDeodorant. Technical report, Department of Computer Science and Software Engineering, Concordia University.

[35] Vojtech Svatek, Ondrej Svab-Zamazal, Miroslav Vacura, Petr Strossa, Marek Dudáš, Ján Cerný, and Simone Serra. PatOMat. Technical report, Knowledge Engineering Group, University of Economics, Prague, Czech Republic.

[36] Andreas Dangel and Romain Pelisse. PMD. Technical report, last update: May 2013.

[37] SonarSource SA. SonarQube. Technical report, last update: June 2013.

[38] Dimitrios Settas. SPARSE (sparse-antipatt). Technical report, last update: April

2013.

[39] John M. Chambers, William S. Cleveland, Paul A. Tukey, and Beat Kleiner. *Graphical Methods for Data Analysis*. Wadsworth International, 1983.

[40] Charles Consel and Renaud Marlet. Architecturing Software Using A Methodology for Language Development. *Lecture Notes in Computer Science*, 1490:170–194, September 1998.

[41] Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais, and Jean-Marc Jézéquel. Evaluation of Kermeta for Solving Graph-based Problems. *Journal on Software Tools for Technology Transfer*, 12(3-4):273–285, July 2010.

[42] David Sciamma, Gilles Cannenterre, and Jacques Lescot. Ecore Tools. Technical report, last update: May 2013.

[43] Obeo. Acceleo. Technical report, 2005.

[44] William B. Frakes and Ricardo A. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.