Detection of SOA Patterns

Anthony Demange, Naouel Moha, and Guy Tremblay

Département d'informatique, Université du Québec à Montréal, Canada
anthonydemange@gmail.com,
{moha.naouel,tremblay.guy}@uqam.ca

Abstract. The rapid increase of communications combined with the deployment of large scale information systems lead to the democratization of *Service Oriented Architectures* (SOA). However, systems based on these architectures (called SOA systems) evolve rapidly due to the addition of new functionalities, the modification of execution contexts and the integration of legacy systems. This evolution may hinder the maintenance of these systems, and thus increase the cost of their development. To ease the evolution and maintenance of SOA systems, they should satisfy good design quality criteria, possibly expressed using patterns. By *patterns*, we mean good practices to solve known and common problems when designing software systems. The goal of this study is to detect patterns in SOA systems to assess their design and their Quality of Service (QoS). We propose a three steps approach called SODOP (Service Oriented Detection Of Patterns), which is based on our previous work for the detection of antipatterns. As a first step, we define five SOA patterns extracted from the literature. We specify these patterns using "rule cards", which are sets of rules that combine various metrics, static or dynamic, using a formal grammar. The second step consists in generating automatically detection algorithms from rule cards. The last step consists in applying concretely these algorithms to detect patterns on SOA systems at runtime. We validate SODOP on two SOA systems: *Home-Automation* and *FraSCAti* that contain respectively 13 and 91 services. This validation demonstrates that our proposed approach is precise and efficient.

Keywords: Service Oriented Architecture, Patterns, Specification and Detection, Software Quality, Quality of Service (QoS), Design.

1 Introduction

Service Oriented Architecture (SOA) is an architectural style increasingly adopted because it offers system architects a high level solution to software design. SOA systems are based upon loosely coupled, autonomous and reusable coarse-grained components called services [22]. Each service provides a domain specific behavior, and services can be composed as composite to fulfill high level business processes requirements. Various technologies have emerged to implement this style, among them, Web Services [14] and SCA [6]. Google, Amazon, Microsoft are well-known businesses that have successfully based their information

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 114–130, 2013.
© Springer-Verlag Berlin Heidelberg 2013

systems on SOA. Software systems evolve rapidly due to the addition of new functionalities and the integration of legacy systems. Well designed systems tend to reduce maintenance effort and costs in the long term [4,21]. However, designing such systems becomes far more complex with the increasing use of distributed and service-based systems. To ease evolution and maintenance, it is important that systems satisfy good design and Quality of Service (QoS) criteria. These concerns were first assessed in the object-oriented (OO) world. For instance, the "Gang of Four" (GoF) [12] proposed several good practices, known as design patterns, to solve common and recurring design problems. In the SOA context, various catalogs [7,9,22] have been published in the last few years to provide similar good patterns to follow. For example, a *Facade*, also referred by the same name in the catalog of OO patterns, correspond to a service that hides complex implementation details. The implementation is decoupled from the service consumer and therefore can evolve independently. A *Router* is another typical SOA pattern [19], which provides an additional layer to service consumers to preclude strong coupling with business services. However, due to their own structural and behavioral properties, SOA and OO patterns remain different.

Various interesting approaches have been proposed to assess software systems quality and efficiency. Many of them focus on automatic design pattern detection in OO systems [3,8,13,15,17]. These are either based on static or dynamic analysis, even sometimes on trace execution mining for architectural style recovery. Thus, they provide a consistent and mature way to assess the quality of OO systems.

Unfortunately, to our knowledge, no such approach exists in the SOA context; that's why we are exploring the SOA patterns detection area. The only closely related work corresponds to our previous work for the detection of SOA antipatterns, which are bad practices by opposition to SOA patterns, which are good practices [20]. A domain specific language provided by the Service Oriented Framework for Analysis (SOFA: `http://sofa.uqam.ca`) allows system analysts to describe bad design practices with a high level expressive vocabulary. Each antipattern, derived from the literature, is specified with rule cards, which are sets of rules that use specific metrics [20]. These can either be static, and thus provide information about structural properties like cohesion or coupling, or dynamic, and provide information about response time or number of service invocation. An automatic generation process converts rule cards into detection algorithms, that can then be applied on the SOA systems under analysis.

In this paper, we extend the existing SOFA framework to consider the detection of SOA patterns at runtime. Until now, no automatic approach for the detection of such patterns has been proposed, making the approach proposed in this paper original. The proposed approach is called SODOP (Service Oriented Detection Of Patterns) and consists in the following three contributions. (1) A thorough domain analysis from different catalogs led us to compile and categorize the best practices in SOA systems and their underlying technologies. (2) This analysis resulted in the specification of five significant SOA patterns using rule cards. We selected these five SOA patterns because they represent

technology-agnostic, common and recurrent good quality practices in the design and QoS of SOA systems. (3) Specifying the appropriate rule cards required us to extend SOFA's existing set of metrics with eight new metrics. We validated the proposed approach with two SOA systems: *Home-Automation*, a system that provides services for domotic tasks, and *FraSCAti*, an implementation of the *Service Component Architecture* (SCA) standard [24]. We show that our SODOP approach allows the specification and detection of SOA patterns with high precision values. More detailed information on our approach and the analyzed systems can be found through the SOFA website (`http://sofa.uqam.ca/sodop`).

Overall, the paper is organized as follows. Section 2 describes related work in SOA patterns and their automatic detection. Section 3 presents the proposed approach for the specification and detection of SOA patterns based on metrics. Section 4 describes experiments and results on the two SOA systems mentioned above. Finally, Section 5 concludes and presents future work.

2 Related Work

Automatic detection of design patterns has already been highly investigated for assessing the quality of OO systems. Antoniol *et al.* proposed one of the first approach for design pattern recovery in OO programs [3]. The first step of this approach consists in mapping source code in an intermediate representation with an abstract object language. In the second step, several static metrics, like the number of attributes, methods or associations, are then computed on this abstract language. The final pattern recognition process is executed by examining relations between classes and matching them with GoF design patterns. However, as in many other work, behavioral patterns were omitted because of the focus on static analyses.

Tsantalis *et al.* proposed an interesting way to recover behavioral patterns through a data-mining process based on execution traces [25]. The process consists in extracting graphs or matrices for each of the following OO concepts: association, generalization, abstract classes and abstract method invocations. Based on design patterns definition from the literature, they identify the best matching results from each matrix and identify candidate patterns. Ka-Yee Ng *et al.* gave an alternative solution based on a dynamic pattern recovery process [18]. They begin with the specification of scenario diagrams for each design pattern to consider. Based on execution traces, the system under analysis is then reverse-engineered based also on a scenario diagram. This program scenario diagram is finally assigned to the initial design pattern scenario diagram with an explanation-based constraint programming to identify potential matches. Wendehals *et al.* combined static and dynamic approaches to recover both structural and behavioral patterns. Their dynamic approach is based on transforming execution calls between objects to finite automata. A matching process between these automata and design patterns templates returns the best patterns candidates.

The majority of the community tends to say SOA was first introduced in 1996 by Schulte and Natiz in their Gartner technical report [23]. SOA patterns

catalogs only appeared starting around 2009 [7,9,22]. Galster *et al.* identified most of the patterns specified in Erl's *SOA Patterns* [9] and showed the positive impacts of patterns on quality attributes [11]. Their approach, manual, consists in specifying quality attributes on each pattern and then identifying them manually in real service-based systems.

Despite the emerging interest in SOA, the literature is not really consistent with respect to SOA pattern definition and specification. Indeed, the available catalogs use different classification, either based on their nature, scope or objectives. After an in depth review, we identified the available patterns and the three main categories in which they fall. The first category describes structural patterns which focus on how services are designed to assess common concerns like autonomy, reuse, or efficiency. The second category represents integration and exchange patterns, and describes how service composition and orchestration are used to answer high level business application needs. This category includes how services communicate with each other using different messaging capabilities like synchronous or asynchronous exchanges. The last category can be seen as specific QoS objective patterns such as scalability, performance or security requirements.

To our knowledge, the only related work investigating design and quality of service-based systems is from Yousefi *et al.* [27]. Their recent work proposed to recover specific features in SOA systems by mining execution traces. By executing specific scenarios provided by a manager, they collect the distributed execution traces. A bottom-up data mining algorithm analyzes the traces to build closed frequent item-sets graphs. A filtering and feature recovering process finally eliminates noises and omnipresent calls. This process allows the extraction of specific scenario features based on call frequency and utilization. The obtained results tend to help maintainers by focusing on the most important service providers to improve the QoS of SOA systems and ease their evolution.

Finally, Hohpe, in his report *SOA Patterns: New Insights or Recycled Knowledge?* [16], explained that SOA is more than "a new fancy technology." It is really a new programming model that requires specific approaches and therefore interests in SOA patterns. Thus, OO software systems cannot be directly compared to SOA systems because they both have their own structural and behavioral properties. Therefore, OO design patterns recovery cannot be directly applied to SOA pattern detection. This is why our approach aims at providing a specific technique to recover SOA patterns in an automated manner.

3 Our Approach SODOP

We propose the SODOP approach (*Service Oriented Detection Of Patterns*) that aims at the specification and automatic detection of SOA patterns. SODOP is an extension of a previous approach proposed by Moha *et al.* [20] called SODA (Service Oriented Detection for Antipatterns). In the following, for the sake of clarity, we first describe the SCA standard key concepts and the SODA approach. Then, we present the SODOP approach and the specification of five SOA patterns as defined with SODOP.

3.1 About the Service Component Architecture

Before introducing the SODA and SODOP approaches, it must be stressed that the following experiments were made with the Service Component Architecture (SCA) standard. A description of the SCA standard and its vocabulary is thus useful to better understand how specific metrics are computed. A software application built with SCA contains one or many components as shown in Figure 1. A component is a logical building block implementing a specific business logic, which is why we consider a component as a high level SOA service in this paper. Each component can expose services, which declare methods potentially called by clients, and references to other services the component depends on. The link between two components is called a wire. A component could potentially nest other components and become a composite. This composite can expose nested components behaviors by promoting their services or references.

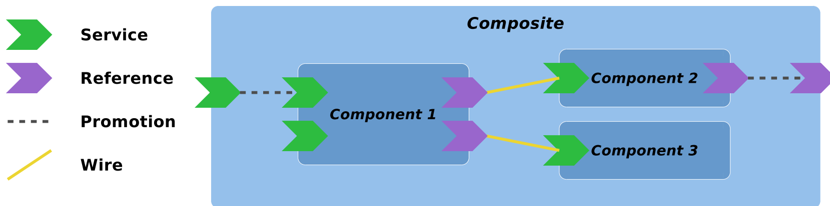

Fig. 1. Key Concepts of the SCA Standard

3.2 Description of the Earlier SODA Approach

SODA proposes a three steps approach for the detection of SOA *antipatterns*—an antipattern corresponds to bad design practices, by opposition to patterns. The first step consists in specifying SOA antipatterns using a Domain Specific Language (DSL) that defines "rule cards", which are set of rules matching specific QoS and structural properties. Figure 3 shows this DSL's grammar, in Backus-Naur Form. A rule describes a metric, a relationship, or a combination of other rules (line 3) using set operators (line 6). A metric can either be static (line 11) or dynamic (line 12)—computed at runtime. Examples of static metrics include number of methods declared (NMD) or number of outgoing references (NOR). Examples of dynamic metrics include response time (RT) or number of incoming calls (NIC). A metric can optionally be defined as an arithmetic combination of other metrics (lines 8 and 9). Each metric can be compared to one ordinal values (line 7)—a five value Likert scale from very low to very high (line 12)—or compared to a numeric value (line 8) using common arithmetic comparators (line 13). A metric value is calculated for each service in the set to populate one box-plot per metric. Figure 2 describes how ordinal values are mapped to box-plot intervals.

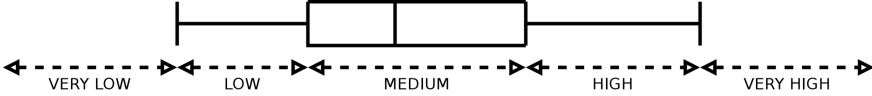

Fig. 2. Mapping between ordinal values and box-plot intervals

The second step consists in generating automatically the detection algorithms corresponding to the rule cards specified. These algorithms were generated with the EMF [2] meta-model combined with the Acceleo [1] code generation tool. The third and final step consists in applying these algorithms on real SOA systems to detect candidate services that match antipattern rule cards. In our case, SCA joint points were woven on each service so that every call trigger an event. Each event is caught so that the computation of metrics is done on the called service.

```
1  rule_card      ::= RULE_CARD:rule_card_name {(rule)+};

2  rule           ::= RULE:rule_name {content_rule};
3  content_rule   ::= metric | set_operator rule_type (rule_type)+
4                     |RULE_CARD: rule_card_name
5  rule_type      ::= rule_name | rule_card_name

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

7  metric         ::= metric_value comparator (metric_value | ordi_value | num_value)
8  metric_value   ::= id_metric (num_operator id_metric)?
9  num_operator   ::= + | - | * | /

10 id_metric      ::= ANAM | ANIM | ANP | ANPT | COH | NID | NIR | NMD | NOR | NSC | TNP
11                    | A | DR | ET | NDC | NIC | NOC | NTMI | POPC | PSC | SR | RT

12 ordi_value     ::= VERY_LOW | LOW | MEDIUM | HIGH | VERY_HIGH
13 comparator     ::= < | ≤ | = | ≥ | >

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

Fig. 3. BNF Grammar for Rule Cards

3.3 Description of the SODOP Approach

The SODA approach is flexible and relatively easy to extend for SOA *patterns* instead of antipatterns. Indeed, the DSL and the underlying SOFA framework allow the integration of new metrics required for the specification of patterns. The approach proposed in this paper, called SODOP, introduces five new patterns, that we identified from the SOA literature. These patterns have been specified with rule cards by combining existing metrics along with eight newly defined ones—those are underlined in Figure 3 and are briefly described below. SODOP's three steps are described in Figure 4, and are similar to SODA's ones. The DSL grammar has been extended to allow more flexibility in the rule card specification. We add the possibility of combining two existing metrics with

numeric operators to avoid the proliferation of new metrics and, thus, to provide ratios. The pattern rule cards specified in Step 1 are generated automatically into detection algorithms in Step 2, followed by the concrete detection of patterns on SOA systems in Step 3. The first specification step is thus manual, whereas the second and third are automated.

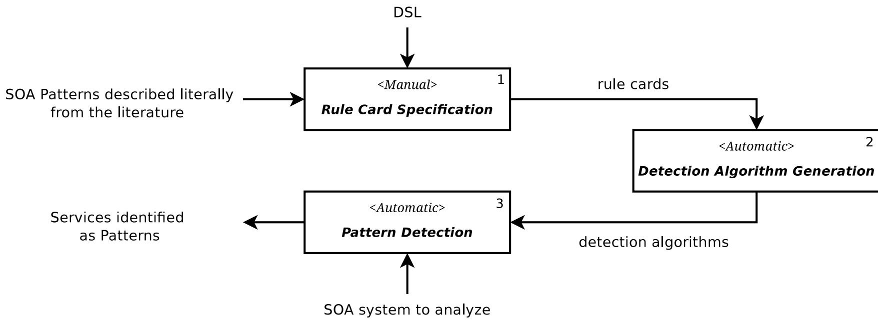

Fig. 4. The Three Steps of the SODOP Approach

The following eight new metrics were defined. The *Execution Time (ET)* represents the time spent by a service to perform its tasks; it differs from the response time as it excludes the execution time of nested services. The *Number of Different Clients (NDC)* is the number of different consumers, thus multiple incoming calls from the same consumer are counted only once. By contrast, the *Number of Incoming Calls (NIC)* and *Number of Outgoing Calls (NOC)* refer to dynamic calls, thus possibly counting several times the same service. The *Delegation Ratio (DR)* represents the ratio of incoming calls that are relayed by a service. The *Service Reuse (SR)* is a dynamic metric that computes to what extent a service is reused; it is the ratio between the incoming calls (NIC) and the total number of calls in the system. The *Proportion of Outgoing Path Change (POPC)* computes the proportion of outgoing paths that change for a given incoming call. In other words, this proportion is zero if the incoming call and its underlying outgoing calls are always the same. Finally, the *Proportion of Signature Change (PSC)* computes the proportion of method signature change for a pair of incoming/outgoing calls. In other words, this metric represents the dissimilarity level between an incoming and outgoing method call; it is computed with the Jaro-Winkler similarity distance between method names [26].

3.4 Basic Service Pattern

When dealing with SOA pattern specification and detection, we want to specify the best fundamental characteristics every system designer or architect should take into account. Several principles, some of which are described in *SOA Patterns* [9], have to be considered for service design. Components reusability (SR) as well as high cohesion (COH) are common requirements in the design of general

systems such as OO systems [5]. The dynamic nature of SOA systems introduces new non-functional requirements such as high availability (A) or low response time (RT). These metrics are combined in the rule card shown in Figure 9(a) for the specification of this *Basic Service* pattern.

3.5 Facade Pattern

A *Facade*, as illustrated in Figure 5, is used in SOA systems to get a higher abstraction level between the provider and the consumer layers. Fowler and Erl describe the pattern respectively as *Remote Facade* [10], *Decoupled Contract* or *Service Decomposition* [9] and give as example using it to wrap legacy systems. This pattern is similar to the *Facade* in OO systems because it hides implementation details [12] such as nested compositions and calls. It also provides loosely coupled relationships with consumer services and let the implementation evolve independently, without breaking the client contract. Using this pattern, it is possible to decompose SOA systems following the principle of *separation of concerns*. It will thus be easier to reuse the different layers in other systems. A *Facade* can be responsible for orchestration, and can describe how composition of subsequent services can fulfill the client requirements. Given that the *Facade* acts as a front layer to several clients, we characterize its response time (RT) as high. Such a pattern is defined to hide implementation details from many services. Thus, its incoming outgoing calling ratio (NIC/NOC) is low because for one incoming call, the components tends to execute multiple outgoing calls. Finally, we assume that such a service has a high delegation ratio (DR) because it does not provide business logic directly but, instead, delegates to other services. Figure 9(b) shows the rule card specification for the *Facade* pattern.

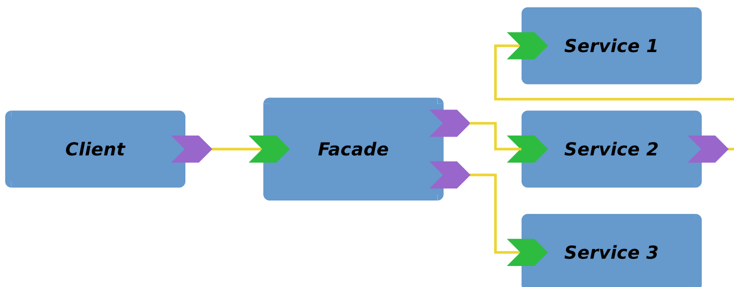

Fig. 5. Facade Pattern Example

3.6 Proxy Pattern

The *Proxy* pattern, represented in Figure 6, is another well-known design pattern from OO systems, that adds an additional indirection level between the client and the invoked service. Its objective differs from a *Facade* because it can, for example, add new non-functional behaviors, which can cover security concerns

Fig. 6. Proxy Pattern Example

such as confidentiality, integrity or logging execution calls for accountability goals. Different kinds of *Proxy* patterns exist, such as *Service Interceptor* [7] or *Service Perimeter Guard* [9], and they could all be specified with several distinct rule cards. Instead, we choose to specify a generic version of this pattern with the following characteristics. The proportion between incoming and outgoing calls (NIC/NOC) has to be equal to one because it acts only as a relay. Moreover, this relay property implies that incoming and outgoing method signatures have to be the same. The fact that the *Proxy* pattern generally adds non-functional requirements to SOA systems also means that it can be involved in several scenarios. Thus, it has a high service reuse (SR) compared to other services. Figure 9(c) shows its underlying rule card.

3.7 Adapter Pattern

The *Adapter* pattern, shown in Figure 7, is also close to the *Adapter* as found in OO systems. Its goal is to adapt the calls between the destination service and the clients. The integration of legacy systems into a SOA system often requires adaptations to perform type transformations and preserve the functionality of the legacy systems. Daigneau gives the example of a *Datasource Adapter* [7] pattern as a solution that provides data access to specific different platforms. In general, the number of incoming and outgoing calls are identical, thus the ratio (NIC/NOC) is equal to one. Given the fact this pattern adapts specific client calls, we can infer a high proportion of signature change (PSC) between incoming and outgoing calls. This characteristic makes the *Adapter* differ from the *Proxy*, which preserves the method signatures and simply relays calls. Figure 9(d) shows the *Adapter* pattern rule card.

Fig. 7. Adapter Pattern Example

3.8 Router Pattern

The *Router* pattern, as illustrated in Figure 8, is similar to a network router that forwards packets according to different paths. A SOA *Router* distributes incoming calls to various destinations based on different criteria, which can be either the client identity or the call parameters. Some smart routers either detect paths on dynamic metrics such as availability or previous calls history and forward calls to the best matching service. The main criterion to consider is a

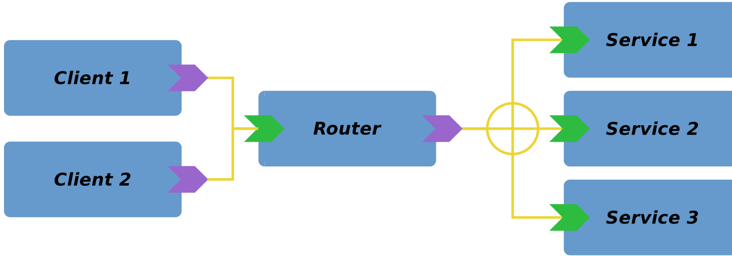

Fig. 8. Router Pattern Example

change of outgoing paths for a specific incoming call, so a high proportion in path changes (POPC) can be significant. It may be interesting to see if some specific clients use specific paths and then make the correlation with incoming parameters. Figure 9(e) shows the *Router* pattern rule card.

```
1 RULE_CARD: Basic Service {
2   RULE: Basic Service {INTER HighSR
3   HighCOH HighA LowRT};
4   RULE: HighSR {SR ≥ HIGH};
5   RULE: HighCOH {COH ≥ HIGH};
6   RULE: HighA {A ≥ HIGH};
7   RULE: LowRT {RT ≤ LOW};
8 };
```

(a) Basic Service

```
1 RULE_CARD: Facade {
2   RULE: Facade {INTER HighDR
3   LowIOCR HighRT};
4   RULE: HighDR {DR ≥ HIGH};
5   RULE: LowIOCR {NIC/NOC ≤ LOW};
6   RULE: HighRT {RT ≥ HIGH};
7 };
```

(b) Facade

```
1 RULE_CARD: Proxy {
2   RULE: Proxy {INTER EqualIOCR
3   HighSR LowPSC};
4   RULE: EqualIOCR {NIC/NOC = 1.0};
5   RULE: HighSR {SR ≥ HIGH};
6   RULE: LowPSC {PSC ≤ LOW};
7 };
```

(c) Proxy

```
1 RULE_CARD: Adapter {
2   RULE: Adapter {INTER EqualIOCR
3   HighPSC};
4   RULE: EqualIOCR {NIC/NOC = 1.0};
5   RULE: HighPSC {PSC ≥ HIGH};
6 };
```

(d) Adapter

```
1 RULE_CARD: Router {
2   RULE: Router {HighPOPC};
3   RULE: HighOPC {POPC ≥ HIGH};
4 };
```

(e) Router

Fig. 9. Rule Cards for SOA Patterns

4 Experiments

To show the usefulness of the SODOP approach, we performed some experiments that consisted in specifying the five SOA patterns presented in the previous section and detecting them automatically on two SCA systems, *Home-Automation*

and *FraSCAti*. *Home-Automation* is a system that provides services for domotic tasks, whereas *FraSCAti* is an implementation of the SCA standard. Concretely, these experiments aim to show the extensibility of the DSL for specifying new SOA patterns, the accuracy and efficiency of the detection algorithms, and the overall correctness of the underlying framework. As part of the experiments, two independent analysts validated results for *Home-Automation* and the FraSCAti team validated the results obtained for their framework. This independent validation enables us to compare the precision and recall of our SODOP approach and demonstrates the accuracy and efficiency of the rule cards and the related detection algorithms.

4.1 Assumptions

The experiments aim at validating the following three assumptions:

A1. Extensibility: *The proposed extended DSL is flexible enough to define SOA patterns.* Through this assumption, we show that although the DSL and the SOFA framework were initially dedicated to the specification and detection of SOA antipatterns, they are sufficiently extensible to handle SOA patterns thorough the use of metrics.

A2. Accuracy: *The services identified as matching our SOA patterns must attain at least 80% of precision and 100% of recall.* We want to guarantee the accuracy and the efficiency of the rule cards and the related detection algorithms by identifying all patterns present in the analyzed systems while still avoiding too many false positives with a high precision value.

A3. Performance: *The time needed by the detection algorithms must not impact the performance of the analyzed system.* We want to keep the detection time required by the SODOP approach and the underlying SOFA framework very low to avoid efficiency issues in the analyzed system.

4.2 Analyzed Systems

The experiments have been performed on two different SCA systems that are in conformance with the SOA principles: *Home-Automation*, composed of 13 services and executed with 7 different scenarios, and *FraSCAti*, an open-source implementation of the SCA standard. *FraSCAti* fully uses SCA service composition as it includes 13 composite components, themselves encapsulating components, for a total of 91 components. The experiment with this system involves the bootstrap and launch of six SCA applications developed within *FraSCAti* to simulate the scenarios.

4.3 Process

The process used for these experiments follows the three steps of the SODOP approach presented in Section 3. We first specified the rule cards representing the five SOA patterns described previously. Then, we generated automatically the detection algorithms in the second step. Finally, we applied them respectively

on *Home-Automation* and *FraSCAti* to detect the SOA patterns specified. We validated the results by computing the precision—the proportion of true patterns in the detected patterns— and the recall—the proportion of detected patterns in all existing patterns. These validations were made through a manual and static analysis of each service in the systems under analysis. The computations were performed by two external software engineers to ensure the results were not biased. An additional feedback was given by the *FraSCAti* core team itself to strengthen the results.

4.4 Results

In the following, we first discuss the results obtained on the two SCA systems. Tables 1 and 2 respectively present the detection results on each system. For each SOA pattern listed in column one, column two describes the services detected as patterns. Columns three, four and five give respectively the value of metrics involved in the rule card of the pattern, the time required for applying the detection algorithms and the system execution time. The two last columns provide the precision and the recall values. The last row gives average values (detection time, execution time, precision and recall).

Details of the Results on *Home-Automation*

Four of the five specified SOA patterns were detected on *Home-Automation*—the *Adapter* pattern was not detected. The *patientDAO*, *communication* and *knxMock* components are detected as *Basic Service* pattern with a maximal cohesion (COH $\geq$ 0.34), high reuse values (SR $>$ 0.10) and very low response time (RT $<$ 0.25ms). According to the definition of the *Basic Service* pattern, these three components thus represent the services in the system that are the most well designed, as they appear to satisfy common software design principles. The *mediator* component is considered both a *Facade* and a *Router*. The *Facade* represents a service acting as a front layer to clients to hide a complex subsystem. Indeed, the delegation metric (DR = 1) of the *mediator* component always acts as a relay and tends to have six times more outgoing calls for each incoming one (NIC/NOC = 0.17), thus this traduces its high response time (RT = 2.8ms). The *mediator* has also been detected as a *Router* because of its high dynamic metric (POPC = 0.5). This value means that the *mediator* distributes to different outgoing paths half of its incoming calls. The *patientDAO* also matches the *Proxy* pattern because of its high reuse (SR = 0.24 compared to the median value of 0.06) and systematic incoming calls relay (NIC = NOC) with the same method signatures (PSC = 0). We can also observe that the time required for the detection of each pattern is on average 25ms, whereas the average execution time on a given set of scenarios is 6.73s. These values demonstrate the low impact of the pattern detection on the system execution, and thus on the results. Finally, the validation performed by the two experts lead to a 93.3% precision and 100% recall, which indicates that all existing patterns in *Home-Automation* have been detected, with high precision.

Table 1. SOA Pattern Detection Results on the Home-Automation System

PatternName	DetectedServices	Metrics			DetectTime	ExecTime	Precision	Recall
		COH	RT	SR				
Basic Service	*patientDAO* *communication* *knxMock*	0.49 0.34 0.38	0.25ms 0.24ms 0.16ms	0.24 0.10 0.11	80ms	6.82s	[3/2] 66.6%	[2/2] 100%
		NIC/NOC	DR	RT				
Facade	*mediator*	0.17	1.0	2.8ms	10ms	6.66s	[1/1] 100%	[1/1] 100%
		NIC/NOC	SR	PSC				
Proxy	*patientDAO*	1.0	0.24	0.0	13ms	6.74s	[1/1] 100%	[1/1] 100%
Adapter	n/a	n/a			10ms	6.76s	[0/0] 100%	[0/0] 100%
		POPC						
Router	*mediator*	0.5			11ms	6.67s	[1/1] 100%	[1/1] 100%
Average					25ms	6.73s	93.3%	100%

Details of the Results on *FraSCAti*
As shown in Table 2, the detection of patterns on *FraSCAti* returns more results than *Home-Automation*, i.e. more components are detected as patterns. This is partly explained by the size of *FraSCAti*, which is almost ten times larger than *Home-Automation*. Five components have been detected as matching the *Basic Service* pattern, because of their very high reusability (SR > 0.1 compared to the median value of 0.003), high cohesion (COH > 0.48) and mostly very low response time (RT < 0.7ms). The core framework components, *FraSCAti*, *assembly-factory* and *composite-parser*, are detected as *Facade* as they are main entry points of the framework that relay every incoming calls (DR = 1). Their incoming outgoing call ratio (NIC/NOC = 0.46, 0.25 and 0.63) remains low compared to the median value of 1. They act as a *Facade* because they have among the highest response times (respectively 571ms, 181ms and 10ms) mainly due to their massive underlying calls. The *Proxy* pattern is involved in the three following components: *sca-interface*, *sca-implementation* and *component-factory*. The components have been identified as *Proxy* because they represent highly reused (SR > 0.5) relay services (NIC/NOC = 1) and they include the same method calls (PSC = 0). The only missing SOA pattern in *Home-Automation* and discovered in *FraSCAti* is the *Adapter*, seen in the *BindingFactory* component. It acts as an *Adapter* because it relays all its incoming calls (NIC/NOC = 1) and adapts the method calls to the underlying components (PSC = 1, which indicates a high proportion of signature change). Unlike in *Home-Automation*, no *Router* pattern has been detected. The average detection time required for our experiments is 97ms on average for a total time average of 10.9s. As for *Home-Automation*, the detection represents only 1% of the total system execution, and thus does not affect its performance, even with a relatively larger system. We reported those results to the *FraSCAti* core team and they confirmed all components detected as patterns. This leads to a precision of 100% for this detection. However, the recall

Table 2. SOA Pattern Detection Results on the FraSCAti System

PatternName	DetectedServices	Metrics	DetectTime	ExecTime	Precision	Recall
Basic Service	*sca-interface* *sca-interface-java* *sca-impl.* *sca-impl.-java* *sca-comp.-service*	COH RT SR 0.48 0.09ms 0.11 0.50 0.02ms 0.11 0.48 0.67ms 0.05 0.50 0.59ms 0.04 0.48 0.25ms 0.48	241ms	11.34s	[5/5] 100%	n/a
Facade	*FraSCAti* *assembly-factory* *composite-parser*	NIC/NOC DR RT 0.46 1.0 571ms 0.25 1.0 181ms 0.63 1.0 10ms	57ms	10.62s	[3/3] 100%	[3/16] 18.7%
Proxy	*sca-interface* *sca-impl.* *component-factory*	NIC/NOC SR PSC 1.0 0.11 0.0 1.0 0.05 0.0 1.0 0.11 0.0	65ms	10.72s	[3/3] 100%	[3/14] 21.4%
Adapter	*BindingFactory*	NIC/NOC PSC 1.0 1.0	57ms	10.96s	[1/1] 100%	[1/14] 7.1%
Router	n/a	n/a	67ms	10.88s	[0/0] 100%	[0/7] 0.0%
Average			97ms	10.90s	100%	11.8%

value of 11.8% is low. Our detection algorithms thus failed at detecting all the existing components involved as patterns in the *FraSCAti* system.

4.5 Discussion

We now discuss the three assumptions mentioned earlier to show the usefulness of the SODOP approach.

A1. Extensibility: *The proposed extended DSL is flexible enough to define SOA patterns.* This first assumption is positively supported because we show through the experiments that the DSL allows designers to define different kinds of rule cards and add new metrics that can be either static or dynamic. Indeed, the specification of SOA patterns required the addition of eight dynamic metrics and the reuse of the 14 existing ones. In addition to the new metrics, the DSL has been extended with numeric operators (`+,-,*,/`) to allow the combination of metrics and, thus, avoid introducing new metric specifications, keeping the language as simple and flexible as possible.

A2. Accuracy: *The services identified as matching our SOA patterns must attain at least 80% of precision and 100% of recall.* The detection results demonstrate the high precision of the SODOP approach, respectively 93.3% and 100% for *Home-Automation* and *FraSCAti*. The recall for *Home-Automation* is 100% but the one for *FraSCAti* is about 12%. This result is related to the highly dynamic detection of patterns, which is based on a set of scenarios that do not cover all the system execution paths. In these experiments with *FraSCAti*, unlike with *Home-Automation*, it is quite difficult to reach 100% coverage because of the system size.

A3. Performance: *The time needed by the detection algorithms must not impact the performance of the analyzed system.* As shown in Tables 1 and 2, no matter which

SOA patterns or how many metrics are computed, the detection time remains low compared to the execution time and thus does not impact the system under analysis. As a first analysis, we find the only affecting property is the number of services involved in the SOA system under analysis, because all the metrics are computed against each of them. *FraSCAti* has around eight times more components than *Home-Automation*, which explains the proportional time needed to run the metrics computation (around 1% of the execution time). Because the experiments are run locally, the execution time is also highly dependent on the computer computational power. In these experiments, an Intel E5345 CPU with 4GB of RAM was used.

4.6 Threats to Validity

Several threats can be considered as counter-measures to the validity of our study. First, the external reliability, i.e., the repeatability of our experiments, is guaranteed under the condition that the same computational facilities are used. This is still guaranteed by the automatic detection algorithms generation, which will be identical for the same input rule card. We provide the details of our results as well as the systems analyzed in the SOFA website (`http://sofa.uqam.ca/sodop`). The main possible external validity threat may come from the fact we only focus on two SCA systems. Although they are representative of small as well as big systems, SOA technologies often have specific characteristics, which is why we plan to extend our study in the future. We tried to minimize the potential construct validity of our approach by providing the most representative execution scenarios for each system under analysis. However for FraSCAti, the scenarios were not exhaustive as highlighted by the recall of 11.8%. Because of the size of the system, we will consider it in our next future experiments. The other construct validity potentially questionable may come from the rule cards subjectivity. Indeed, this depend heavily on the designer specifying them, but we tried as much as possible to remain close and faithful to the SOA patterns described in the literature. Moreover, although we only defined five SOA patterns in the form of rule cards, they are representative according to the literature. Indeed, even if SOA catalogs mainly define patterns for specific technologies, we tried to specify meaningful technology-agnostic SOA patterns.

5 Conclusion and Future Work

SOA patterns are proven good practices to solve known and common problems when designing software systems. Indeed, our three steps SODOP approach consists in the specification and detection of SOA patterns to assess the design and QoS of SOA systems. The first step consists in specifying rule cards—set of rules, combining static and dynamic metrics—for each pattern. Five patterns were described in our study, involving 22 different static and dynamic metrics, including eight newly defined dynamic metrics. The second step consists in generating automatically detection algorithms from rule cards, and applying them on SOA systems in the third step. We validated our approach using two SCA systems, *Home-Automation*—a system that provides 13 services for domotic tasks—and

FraSCAti—a SCA standard implementation that provides 91 components. The experiments showed that we can obtain high precision and recall values under the condition that execution scenarios are exhaustive.

Various lines of future work are currently being explored by our research group. First, we will expand the SODOP approach by specifying more SOA patterns and applying them on other SOA systems. We also plan to extend our approach to other SOA technologies, such as Web Services and REST, as they share many common properties. Our approach remain however applicable to these other technologies to the condition we wrap them in specific SCA containers.

Acknowledgments. The authors would like to thank the FraSCAti core team, and in particular Philippe Merle, for the validation of the results on FraSCAti and their valuable discussions on these results. This work was partially supported by *Research Discovery* grants from NSERC (Canada).

References

1. Acceleo code generator tool, http://www.acceleo.org/
2. Eclipse modeling framework project, http://www.eclipse.org/modeling/emf/
3. Antoniol, G., Fiutem, R., Cristoforetti, L.: Design Pattern Recovery in Object-Oriented Software. In: 14th IEEE Intl. Conf. on Prog. Comprehension, pp. 153–160 (June 1998)
4. Banker, R.D., Datar, S.M., Kemerer, C.F., Zweig, D.: Software complexity and maintenance costs. Comm. of the ACM 36(11), 81–94 (1993)
5. Basili, V., Briand, L., Melo, W.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 22(10), 751–761 (1996)
6. Chappell, D.: Introducing SCA (2007), http://www.davidchappell.com/articles/introducing_sca.pdf
7. Daigneau, R.: Service Design Patterns. Addison-Wesley (2011)
8. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Improving Behavioral Design Pattern Detection through Model Checking. In: 14th European Conf. on Soft. Maintenance and Reengineering, pp. 176–185. IEEE Comp. Soc. (March 2010)
9. Erl, T.: SOA Design Patterns. Prentice Hall PTR (2009)
10. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
11. Galster, M., Avgeriou, P.: Qualitative Analysis of the Impact of SOA Patterns on Quality Attributes. In: 12th Intl. Conf. on Quality Software, pp. 167–170. IEEE (August 2012)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
13. Guéhéneuc, Y.G., Antoniol, G.: DeMIMA: A Multilayered Approach for Design Pattern Identification. IEEE Trans. on Soft. Eng. 34(5), 667–684 (2008)
14. Hansen, M.D.: SOA Using Java Web Services. Prentice Hall (2007)
15. Heuzeroth, D., Holl, T., Hogstrom, G., Löwe, W.: Automatic design pattern detection. In: Intl. Symp. on Micromechatronics and Human Science, pp. 94–103. IEEE Comp. Soc. (2003)

16. Hohpe, G., Easy, C.: SOA Patterns New Insights or Recycled Knowledge. Tech. rep. (2007)
17. Hu, L., Sartipi, K.: Dynamic Analysis and Design Pattern Detection in Java Programs. In: 20th Intl. Conf. on Soft. Eng. and Data Eng., pp. 842–846 (2008)
18. Ka-Yee Ng, J., Guéhéneuc, Y.G., Antoniol, G.: Identification of Behavioral and Creational Design Patterns through Dynamic Analysis. In: 3rd Intl. Work. on Progr. Comprehension through Dynamic Analysis, pp. 34–42. John Wiley (2007)
19. Milanovic, N.: Service Engineering Design Patterns. In: Second IEEE Intl. Symp. on Service-Oriented System Eng., pp. 19–26 (October 2006)
20. Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.-G., Baudry, B., Jézéquel, J.-M.: Specification and Detection of SOA Antipatterns. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 1–16. Springer, Heidelberg (2012)
21. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Proc. Conf. on Soft. Maint., pp. 337–344. IEEE Comp. Soc. Press (1992)
22. Rotem-Gal-Oz, A.: SOA Patterns. Manning Publications (2012)
23. Schulte, R.W., Natis, Y.V.: Service Oriented Architectures, Part 1. Tech. rep., Gartner (1996)
24. Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable SCA Applications with the FraSCAti Platform. In: 2009 IEEE Intl. Conf. on Services Computing, pp. 268–275. IEEE Computer Society (September 2009)
25. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.: Design Pattern Detection Using Similarity Scoring. IEEE Trans. on Soft. Eng. 32(11), 896–909 (2006)
26. Winkler, W.E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage (November 1989)
27. Yousefi, A., Sartipi, K.: Identifying distributed features in SOA by mining dynamic call trees. In: 27th IEEE Intl. Conf. on Soft. Maint., pp. 73–82. IEEE (September 2011)