Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces

Mathieu Nayrolles, Naouel Moha, and Petko Valtchev
LATECE Team, Département d'informatique, Université du Québec à Montréal, Canada
mathieu.nayrolles@gmail.com, {moha.naouel,valtchev.petko}@uqam.ca

***Abstract*—Service Based Systems (SBSs), like other software systems, evolve due to changes in both user requirements and execution contexts. Continuous evolution could easily deteriorate the design and reduce the Quality of Service (QoS) of SBSs and may result in poor design solutions, commonly known as SOA antipatterns. SOA antipatterns lead to a reduced maintainability and reusability of SBSs. It is therefore important to first detect and then remove them. However, techniques for SOA antipattern detection are still in their infancy, and there are hardly any tools for their automatic detection. In this paper, we propose a new and innovative approach for SOA antipattern detection called SOMAD (Service Oriented Mining for Antipattern Detection) which is an evolution of the previously published SODA (Service Oriented Detection For Antpatterns) tool. SOMAD improves SOA antipattern detection by mining execution traces: It detects strong associations between sequences of service/method calls and further filters them using a suite of dedicated metrics. We first present the underlying association mining model and introduce the SBS-oriented rule metrics. We then describe a validating application of SOMAD to two independently developed SBSs. A comparison of our new tool with SODA reveals superiority of the former: Its precision is better by a margin ranging from 2.6% to 16.67% while the recall remains optimal at 100% and the speed is significantly reduces (2.5+ times on the same test subjects).**

***Index Terms*—SOA Antipatterns, Mining Execution Traces, Sequential Association Rules, Service Oriented Architecture.**

I. Introduction

Service Based Systems (SBSs) are composed of ready-made services that are accessed through the Internet [1]. Services are autonomous, interoperable, and reusable software units that can be implemented using a wide range of technologies like Web Services, REST (REpresentational State Transfer), or SCA (Service Component Architecture, on the top of SOA.). Most of the world's biggest computational platforms: Amazon, Paypal, and eBay, for example, represent large-scale SBSs. Such systems are complex–they may generate massive flows of communication between services–and highly dynamic: services appear, disappear or get modified. The constant evolution in an SBS can easily deteriorate the overall architecture of the system and thus bad design choices, known as SOA *antipatterns* [2], may appear. An antipattern is the opposite of a design pattern: while patterns should be followed to create more maintainable and reusable systems [3], antipatterns must be avoided since they have a negative impact, e.g., hinder the maintenance and reusability of SBSs.

Given their negative impact, there is a clear and urgent need for techniques and tools to detect SOA antipatterns. Recently, a tool was developed by our team, called SODA (Service Oriented Detection for Antipatterns) [2], [4], which targets SOA antipatterns. The tool employs a Domain Specific Language (DSL) to specify SOA antipatterns, which is based on metrics and generates detection algorithms from antipattern specifications in an automated way.

Albeit efficient and precise, SODA suffers from serious limitations. Indeed, the tool performs two phases of analysis, first, a static one and then a dynamic one. The static analysis requires access to service interfaces. Consequently, SODA cannot analyze systems that are proprietary or not open-source. The dynamic analysis requires the execution of the system and therefore, the creation of runnable scenarios. Moreover, since SODA specifically targets systems implementing the SCA standard, its precision drops as the target system gets bigger. Given these limitations, there is a space for improvement, both in precision and in coverage, i.e., detection of antipatterns in SBSs implementing a wider range of SOA technologies. In this article, we propose a new and innovative approach for the detection of SOA antipatterns named SOMAD (Service Oriented Mining for Antipattern Detection). SOMAD does not require scenarios to concretely invoke service interfaces as it only relies on execution traces (provided by any SOA technology). It discards irrelevant data by using data mining techniques–sequential association rules mining–. The tool discovers SOA antipatterns by first extracting associations between services as expressed in the execution traces of an SBS. To that end, it applies a specific variant of the association rule mining task based on sequences or episodes: In our case the sequences represent service or, alternatively, method calls. Further on, generated association rules are filtered using a suite of dedicated metrics.

We applied SOMAD on two different SBS called *Home Automation* and *FraSCAti* [5]. *Home Automation* is made of 13 services and *FraSCAti* is almost ten times larger. We compared the outcome of SOMAD to the one produced by SODA, the so far unique tool for SOA antipatterns detection from the literature. Both tools were evaluated in terms of precision and recall, on one hand, and efficiency, on the other hand. The study results indicate that SOMAD significantly outperforms SODA in term of precision (2.6% to 16.67%) and efficiency (2.5+ times faster).

The main contribution of this paper is thus twofold: (i) a new approach for the detection of SOA antipatterns based on association rules mining from the execution traces of an SBS (from a variety of SOA technologies); (ii) an empirical

978-1-4799-2931-3/13/$31.00 © 2013 IEEE

validation of this approach, which shows the tool outperforms its direct competitor in terms of precision and efficiency.

The remainder of the article is organized as follows. Section II presents related works on pattern and antipattern detection both in SOA and OO paradigms and related works on knowledge extraction. Section III presents the SOMAD approach, and in particular the mining of association rules from execution traces while, Section IV presents our experimental study with a comparison of our SOMAD approach to SODA. Finally, we provide some concluding remarks in Section V.

II. Related Work

As our approach combines antipattern detection and knowledge extraction from execution traces we provide short serveys of related work on both: Section II-A deals with detection of patterns and antipatterns both in OO and SOA paradigms while Section II-B addresses knowledge extraction. Finally, Section II-C presents the SODA approach [2], [4].

A. Pattern and Antipattern Detection

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, and thus target high quality in systems. A number of methods and tools exist for the detection of antipatterns in OO systems [6], [7], [8] whereas the relevant theory and practices have been summarized in best-sellers books [9], [10]. However, the detection of SOA antipatterns, unlike their OO counterparts, is still in its infancy.

An approach to the declarative specification of antipatterns, called SPARSE, is presented in [11]. In SPARSE, antipatterns are described as an OWL ontology augmented with a SWRL (Semantic Web Rule Language) rule basis whereas their occurrences are tested through automated reasoning.

Other relevant work has focused on the detection of specific antipatterns related to the system's performance and resource usage and/or given technologies. For example, Wong *et al.* [12] use a genetic algorithm for detecting software faults and anomalous behavior in the resource usage of a system (e.g. memory usage, processor usage, thread count). The approach is driven by *utility functions* that correspond to predicates identifying suspicious behavior by means of resource usage metrics. In another related work, Parsons *et al.* [13] tackled the detection of performance antipatterns. They use a rule-based approach made of both static and dynamic analyzes that are tailored to component-based enterprise systems (in particular, JEE applications).

B. Knowledge Extraction

A large number of studies focused on knowledge extraction from execution traces. They were motivated by the identification of: crosscutting concerns (aspects) [14], business processes [15], patterns of interests among service users [16], [17], and features either in OO systems [18] or SBSs [19]. Further related work focused on the identification of service composition patterns [20], i.e. sets of services that are repetitively used together in different systems and that are structurally and functionally similar. Composition patterns embody good practices in designing and developing SBSs.

Few projects have explored pattern detection through execution trace mining. Ka-Yee Ng *et al.* [21] proposed MoDeC, an approach for identifying behavioral and creational design patterns using dynamic analysis and constraint programming. They reverse-engineer scenario diagrams from an OO system by bytecode instrumentation and apply constraint programming to detect these patterns as runtime collaborations. Hu and Sartipi [22] tackle the detection of design patterns in traces using scenario execution, pattern mining, and concept analysis. The approach is guided by a set of feature-specific scenarios to identify patterns, as opposed to a general pattern detection.

Although different in goals and scope, the above studies on OO antipatterns form a sound basis of expertise and technical knowledge for building methods for the detection of SOA antipatterns. However, despite a large number of commonalities, OO (anti)pattern detection methods cannot directly apply to SOA. Indeed, SOA focuses on services as first-class entities and thus remains at a higher granularity level than OO classes. Moreover, the highly dynamic nature of a SBS raises challenges that are not preponderant in OO systems.

C. SODA : The State-of-the-Art Tool

SODA relies on a rule-based language that enables antipatterns specification using a set of metrics. A generic process then turns the specification into detection algorithms. The three main steps of the processing are as follows (see Figure 1):

Specification of SOA Antipatterns: Relevant properties of SOA antipatterns are identified, which essentially correspond to metrics such as cohesion, coupling, number of methods, response time and availability. These properties compose to a base vocabulary of a DSL: a rule-based language is used whereby each rule expresses tendencies in metric values. An antipattern is described by a set of rules combined into a *rule card*.

Generation of Detection Algorithms: Automatic generation of detection algorithms is performed by visiting models of rule cards specified during the previous step. The process is straightforward and ends up with a set of directly executable algorithms.

Detection of SOA Antipatterns: The detection algorithms generated in the previous step are applied on the SBS of interest. This step allows the automatic detection of SOA antipatterns using a set of predefined scenarios to invoke service interfaces. At the end, services from the SBS suspected of being involved in an antipattern are identified.

Although efficient and precise, SODA is an intrusive approach because it requires a set of valid scenarios concretely invoking the interface methods of SBSs and its dynamic analysis involves SCA properties.

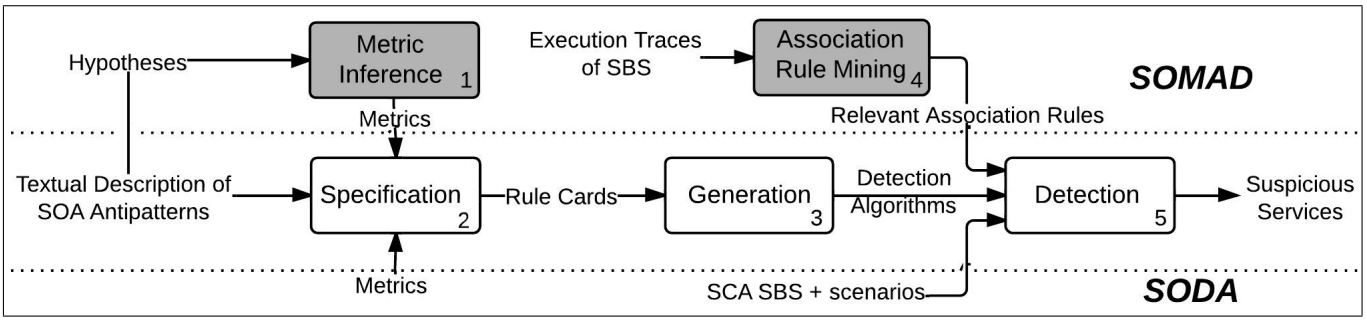

Fig. 1: SODA and SOMAD approaches: Grey boxes depict new steps in SOMAD w.r.t. to SODA (white boxes).

III. The SOMAD Approach

We propose a five step approach, named SOMAD (Service Oriented Mining for Antipatterns Detection), for the detection of SOA antipatterns within execution traces of SBSs. This new approach is a variant of SODA based on execution traces, which may come from any kind of SBSs. In contrast, SODA applies specifically on SCA SBSs using a set of scenarios and SCA-based techniques. In particular, in SOMAD, we specify a new set of metrics that apply to sequential association rules mined on execution traces whereas, in SODA, metrics apply to the concrete invocation of SBSs' interfaces using a set of scenarios. Figure 1 shows an overview of SOMAD. We emphasized in grey the two new steps specific to SOMAD and added to the SODA approach. *Step 1. Metric Inference* is supported by the creation of a set of hypotheses made from the textual description of SOA antipatterns. The hypotheses underlie the definition of new metrics to support the interpretation of association rules. *Step 4. Association Rule Mining* (ARM) discovers interesting sequential associations in execution traces of the targeted SBS. Output sequential association rules represent statistically interesting relations between services inside traces. In what follows, we first introduce key concepts of sequential ARM and then, present the overall process of SOMAD. Finally, we provide some implementation details.

A. Introduction to Sequential Association Rule Mining

In the data mining field, ARM is a well-established method for discovering co-occurrences between attributes in the objects of a large data set [23]. Plain associations have the form $X \rightarrow Y$, where X and Y, called the *antecedent* and the *consequent*, respectively, are sets of descriptors (purchases by a customer, network alarms, or any other general kind of events). Even though plain association rules could serve some relevant information, we are interested here in the sequences of service invocations. We therefore adopt a variant called sequential association rules in which both X and Y become sequences of descriptors. Moreover, our sequences follow a temporal order with the antecedent preceding the consequent. Rules of this type mined from traces reveal crucial information about the likelihood that services appear together in an execution trace and, more importantly, in a specific order. For instance, a strong rule *ServiceA* , *ServiceB* implies *ServiceC* would mean that after executing A and then B, there are good chances to see C in the trace. The conciseness of this example should not confuse the reader as in practical cases the sequences appearing in a rule can be of an arbitrary length. Furthermore, the strength of the rule is measured by the *confidence* metric: In probabilistic terms, it measures the conditional probability of C appearing down the line. Beside that, the significance of a rule, i.e. how many times it appears in the data, is provided by its *support* measure. To ensure only rules of potentially high interestingness are mined, the mining task is tuned by minimal thresholds to output only the sufficiently high scores for both metrics.

B. SOMAD Process

Step 1. Metrics Inference: Metrics to support the interpretation of sequential association rules are inferred from a set of three hypotheses synthesized from the textual description of SOA antipatterns (Table I).

These hypotheses represent heuristics that enable the identification of architectural properties relevant to SOA antipatterns. Indeed, after a careful examination of the textual descriptions, we observed that SOA antipatterns can be specified in terms of coupling and cohesion[1].

Hypothesis 1. *If a service A* implies *a service B with a high support and a high confidence, then A and B are tightly coupled.*

Hypothesis 2. *If a service appears in the consequent (antecedent) parts for a high number of associations, then it has high incoming (outgoing) coupling.*

The above hypotheses qualify the coupling between two specific services and overall incoming/outgoing coupling. The cohesion is also widely used in SOA antipattern descriptions.

Hypothesis 3. *If the number of different methods of a service A is equal or superior to the number of different services invoking A (Hypothesis 2) then, the service is not externally cohesive.*

This definition of cohesion has been introduced by Perepletchikov *et al.*: "*A service is deemed to be Externally*

[1]Recall coupling basically refers to the degree a services relies on others while cohesion measures the relatedness between its own responsibilities [24].

TABLE I: List of SOA Antipatterns [2]

Multi-Service, *a.k.a* God Object corresponds to a service that implements a **multitude of methods** related to different business and technical abstractions. This aggregate too much into a single service, such a service is not easily reusable because of the **low cohesion** of its methods and is often unavailable to end-users because of its overload, which may induce a high response time [25].

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 [25].

Chatty Service corresponds to a set of services that exchange a **lot of small data** of primitive types. The Chatty Service is also characterized by a **high number of method invocations**. Chatty Service chats a lot with each other [25].

The Knot is a **set of very low cohesive** services, which are tightly coupled. These services are thus less reusable. Due to this complex architecture, the availability of these services can be low, and their response time high [26].

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 higher 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 in OO systems, corresponds to a **chain of services**. The Service Chain appears when clients request **consecutive service invocations** to fulfill their goals. This kind of **dependency chain** reflects the action of invocation in a transitive manner.

```
1 RULE_CARD: MultiService {
2  RULE: MultiService{INTER LowCohesion ManyMethods ManyMatches};
3  RULE: LowCohesion{COH LOW};
4  RULE: ManyMethods{NM HIGH};
5  RULE: ManyMatches{NMA HIGH};
6 };
```
(a) Multi Service

```
1 RULE_CARD: TinyService {
2  RULE: TinyService{INTER HighOutgoingCoupling FewMethods};
3  RULE: HighOutgoingCoupling{OC HIGH};
4  RULE: FewMethods{NM LOW};
5 };
```
(b) Tiny Service

```
1 RULE_CARD: ChattyService {
2  RULE: ChattyService{INTER ManyPartners ManyMatches};
3  RULE: ManyPartners{NDP VERY HIGH};
4  RULE: ManyMatches{NMA VERY HIGH};
5 };
```
(c) Chatty Service

```
1 RULE_CARD: BottleNeck {
2  RULE: BottleNeck{INTER HighOutgoingCoupling HighIncomingCoupling};
3  RULE: HighOutgoingCoupling{OC HIGH};
4  RULE: HighIncomingCoupling{IC HIGH};
5 };
```
(d) BottleNeck Service

```
1 RULE_CARD: KnotService {
2  RULE: KnotService{INTER LowCohesion HighCrossInvocation};
3  RULE: LowCohesion{COH LOW};
4  RULE: HighCrossInvocation{CID HIGH};
5 };
```
(e) Knot Service

```
1 RULE_CARD: ServiceChain {
2  RULE: ServiceChain{HighTransitiveCoupling};
3  RULE: HighTransitiveCoupling{TC HIGH};
4 };
```
(f) Service Chain

Fig. 2: Rule Cards

cohesive when all of its service operations are invoked by all the clients of this service" [27]. Based on the above three hypotheses, we have created domain specific metrics to help us explore the antipattern manifestations that are hidden in the sequential association rules. We use the DSL we defined in [2] to combine them. Metrics are presented in Table II. In the figure, standard mathematical notations are used whenever possible and extended if necessary. Thus, association rules are visualized by (X $\rightarrow$ Y) with X and Y represent the antecedent and the consequent parts, respectively. *K, L* are partner services. AR stands for the overall set of association rules while AR_s and AR_m being subsets targeting association rules at service / method level, respectively. M_S denotes the methods of a given service S. Finally, we use non-standard symbols for sequence operations: $[]$ is the sequence constructor, $\Cup$ stand for append on sequences; $\Subset$ denotes the sub-sequence-of relationship; and A $\lessdot$ B means the service/method A appears inside the association rule B. Metrics can be combined to define other metrics.

Step 2. Specification of SOA Antipatterns: The combination of metrics defined in the previous step allows the specification of SOA antipatterns in the form of sets of rules, called *rule cards*.

For the individual metrics and combinations thereof, the values that trigger the detailed examination of a case are not fixed beforehand. Instead, we use a boxplot-based statistical technique that exploits the distribution of all values across the sets of services, methods, and rules. Moreover, the computed values are further weighted using the quality metrics for associations, i.e. support and confidence, so that the strongest rules could be favored. The *rule cards* used to specify SOA antipatterns are presented in Figure 2. As an example, the rule card corresponding to the Tiny Service specification (Figure 3(b)) is composed of three rules. The first one (line 2) is the intersection of two rules (lines 3, 4), which define two metrics: a high Outgoing Coupling (OC) and a low Number of Method (NM).

Step 3. Generation of Detection Algorithms: This step stays unchanged from SODA, as described in Section II-C.

Step 4. Association Rule Mining: Execution traces are analyzed to extract the sequential association rules.

Association rules are extracted from a collection of sequence-shaped transactions with respect to a minimal support and a minimal confidence threshold. A transaction is a time-ordered set of different services and method calls. Recall that the support of a pattern, i.e. sequence of items (services or service methods), reflects the overall percentage of transactions that contain the pattern, whereas the confidence measures the likelihood of the consequent following the occurrence of the antecedent in a transaction. For our experiments (see next section) we set the values of the thresholds to 40% and 60%, respectively. The choice of these values does not follow any specific indication, general law from ARM or deeper insight into the SBS architecture. As our approach is at its exploratory stage, we were only guided by the need to filter out all spurious associations while still keeping enough rules to represent the

TABLE II: Metrics $\Cup$: append on sequences; $\Subset$: sub-sequence-of relationship; and A $\lessdot$ B : A appears inside B.

Number of Matches (NMA(S)) : $\#\{X \to Y \in AR_s \mid S \lessdot (X \Cup Y)\}$ Follows the number of rules where a service appears, either on the left- or on the right-hand side.
Number of Diff. Partners (NDP(S)) : $\#\{K \mid X \to Y \in AR_s, S \lessdot X, K \lessdot Y\} + \#\{K \mid X \to Y \in AR_s, S \lessdot Y, K \lessdot X\}$ Indicates how many different partners a service has. Spelled differently, the metric determines whether the service communicates intensively with surrounding services or not.
Number of Methods (NM(S)) : $\#\{K \mid X \to Y \in AR_m, K \in M_s, K \lessdot (X \Cup Y)\}$ Counts the number of occurrences of the methods from a service. The counting for this metric focuses on method rules.
Cohesion (COH(S)) : $\frac{NDP(S)}{NM(S)}$ Assesses the ratio between the numbers of partner services and of the available methods, respectively.
Cross Invocation Dependencies (CID(S_a, S_b)) : $\#\{X \to Y \in AR_s \mid S_a \lessdot X, S_b \lessdot Y\} + \#\{X \to Y \in AR_s \mid S_a \lessdot Y, S_b \lessdot X\}$ CID is a keystone of the SOMAD approach. Indeed, the metric would explore the typical interactions between services while ignoring less frequent ones (absent from the mining method output due to the support threshold). To retrieve this information CID counts all association rules where a service A (Sa) is present in the antecedent and a service B (Sb) in the consequent or *vice versa*.
Incoming Coupling (IC(S)) : $\sum_{L \in \{K \mid X \to Y \in AR_s, K \lessdot X, S \lessdot Y\}} \frac{CID(S,L)}{NDP(S)}$ Counts how many times a service is used. Yet instead of merely counting a unit for each partner service, we use a contextual value: $\frac{CID(S,X)}{NDP(S)}$ where X is the partner service. Thus, the larger the portion of the partner service in the overall number of partners of S, the higher the coupling.
Outgoing Coupling (OC(S)) : $\sum_{L \in \{K \mid X \to Y \in AR_s, S \lessdot X, K \lessdot Y\}} \frac{CID(L,S)}{NDP(S)}$ The same principle as for **IC**, yet applied in a dual manner: counts how many times the argument service uses other ones.
Transitive Coupling (TC(S_a, S_b)) : $\#\{K \mid X \to Y \in AR_s, S_a \lessdot X, S_b \lessdot Y, ([S_a, K] \Subset X \vee [K, S_b] \Subset Y)\}$ Metric targets the *Service Chain* SOA antipattern (see above). First, observe that the founding idea of *Service Chain* is that absence of direct communication between a pair of services does not mean zero coupling. To identify transitive coupling manifestations we need to capture the notion of a chain: e.g. a service S_a is in the antecedent of a rule, another one S_b is in the consequent of another rule and both rules are connected by means of a third service K that appears in the consequent of the first rule and in the antecedent of the second one. Longer chains are possible as well. Thus, in the basic case, one could have [A]→[B] and [B]→[C]. In this configuration, although A and C are not directly coupled, if C fails, there are good chances that A (and B) would fail too.

most significant calls (regulated via the support threshold). Moreover, we needed enough confidence in the threshold to make appear the most significant alternatives (rule consequent) for the termination of a specific sequence of calls (rule antecedent) while suppressing the less significant ones. Thus, we have made several incremental attempts, starting from 10% and 40% respectively for the support and the confidence. For each attempt, we modified one of the two values by 5% and observed the number of generated rules. The current values seem to offer the best trade-off between size and completeness of scenarios. Now we faced a two-fold possibility for the effective ARM method to use on our traces. In fact, most sequential pattern mining and ARM algorithms have been designed for structures that are slightly more general than ours, i.e. involving sequences of *sets* (instead of single items). Efficient sequential pattern/rule miners have been published, e.g. the PrefixSpan method [28]. In contrast, execution traces do not compile to fully-blown sequential transactions as the underlying structures are mere sequences of singletons, a data format known for at least 15 years yet rarely exploited by the data mining community, arguably because it is less challenging to mine. However, many practical applications have been reported where such data arise, inclusive software log mining (see Section II). In the general data mining literature, mining from pure sequences, as opposed to sequences made of sets, has been addressed under the name of episode mining [29]. Episodes are made of *events* and in a sense, service calls are events. Arguably the largest body of knowledge on the subject belongs to the web usage mining field: The input data is again a system trace, yet this time the trace of requests sent to a web server [30]. Since sequential patterns are more general than the pure sequence ones, mining algorithms designed for the former might prove to be less efficient when applied to the latter (as additional steps might be required for listing all significant sets). Nevertheless, to jump-start our experimental study and given the specificity of our datasets, we choose the RuleGrowth algorithm [31] that seemed to fit at best. Although it has not been optimized for pure sequences its performances are more than satisfactory. In summary, at the end of this, we have extracted the statistically relevant relationship between services in the form of sequential association rules.

Step 5. Detection of SOA Antipatterns

The last step of SOMAD applies the detection algorithms generated in Step 3 to the sequential association rules mined in Step 4. At the end of this step, services in the SBS suspected of being involved in an antipattern are identified and stored for further examination.

C. Implementation Details

In this subsection, we present implementation details for other steps that may support the SOMAD approach.

Generation of Execution Traces. In case execution traces are not available, this step allows their generation.

If the target SBS does not produce qualitative execution traces that contain all the required information, we have to instrument it. Thus, SOMAD requires either the its source code or the execution environment. In fact, such traces enable low-tech application debugging support whenever debuggers are unavailable or inapplicable (frequently the case with SOA environments). Therefore, even if trace producing can introduce source code obfuscation, it may nevertheless have some secondary benefits e.g. in terms of design quality as the code must be well mastered in order to correctly instrument. This technique of tracing is the most common. If the source code is unavailable an alternative consists in instrumenting the running environment of the SBS, i.e. the virtual machine, the web server, or the operating system. For example, LTTng [32] instruments Linux to produce traces with a very low overhead.

To ease automated processing of traces, we provide a template (see Figure 3) that is a good trade-off between simplicity and information content. In this template, a method invocation generates two lines, an opening and a closing one with belonging customer identification (IP address) and a timestamp.

```
IP timestamp void methodA.ServiceA();
IP    timestamp void methodB.ServiceB();
IP    timestamp end void methodB.ServiceB();
IP timestamp end void methodA.ServiceA();
```

Fig. 3: Trace template

Collecting and Aggregating Traces. The goal here is to download all distributed trace files and merge them into a single one.

Traces are typically generated by a set of services within the SBS. Their collection and aggregation is a key yet non-trivial task [33]. Indeed, the dynamic and distributed nature of SBSs is the origin of some serious challenges. One of them is related to the distribution of SBSs and, hence, of execution traces. In fact, each service will generate its execution traces in its own running environment. Therefore, we need to know the name and running place for each service and to have a mechanism for download / retrieval of execution traces on each running environment. Moreover, services can be consumed by several customers simultaneously, hence execution traces can be interleaved. To solve these problem we applied an approach inspired by A. Yousefi and K. Startipi [34]: We first gather all executions log files in one file. Then, we sort execution traces using their timestamps and exploit the caller-callee relationships determined by service and method names to identify blocks of concurrent traces.

Focus shift. This feature is the main reason for SOMAD performing better than SODA in the identification of truly harmful SOA antipatterns.

Observe that SOMAD hypotheses shift the focus of the antipattern search from pure architectural considerations to usage, thus neglecting the exact values of some basic metrics. It is a natural choice since SOMAD does not access exact values through service interfaces or implementation. Moreover, analyzing a system from the usage view angle should –and this was proven by our experimental study (see below)– result in a better precision. Consider a service named *Half-Deprecated Service* composed of four methods: A, B, C and D. Assume the methods C and D are outdated yet the service still exposes them to ensure retro-compatibility. One way to compute the cohesion of our service is to count how many of its methods are used during a session by a unique user. Since half of the methods are outdated it is highly probable that any user will consume at most the other half. Therefore, if cohesion is computed from the service interface, it would amount to 0.5 (2/4) which should raise the suspicions of low-cohesion SOA antipatterns. In contrast, if the cohesion is computed from execution traces the result will tend to be 1.0. Indeed, the unforeseen calls to the deprecated methods will most probably be discarded due to their their low support in the execution traces. In summary, because of its focus on usage, SOMAD should perform better than SODA in detecting harmful SOA antipatterns.

IV. Experiments

As a validation study, we apply SOMAD on two independently developed SBSs, *Home Automation* and *FraSCAti* [5]. *Home Automation* is an SBS made of 13 services and selected for comparison with the outcome produced by SODA, the so far unique state-of-the-art tool for antipatterns detection. Both tools were evaluated in terms of precision and recall, on one hand, and efficiency, on the other hand. We also apply SOMAD to *FraSCAti*, an SBS almost 10 times larger than *Home Automation*, which contains 91 components and 130 services.

A. Subjects

We apply SOMAD to detect six different SOA antipatterns described in Table I. In the description of each antipattern, we highlight in bold the characteristics relevant for their detection using our metrics.

B. Objects

A first round of experiments was performed on *Home Automation*, the same system used in the validation of SODA. *Home Automation* is an independently developed SCA-based system for remotely controlling basic household functions (i.e., temperature, electrical instruments, medical emergency support, etc.) in home care support for elderly. It also includes a set of 7 predefined scenarios for test and demonstration purposes. Two different versions of the system were used: the original version, made of 13 services, and an intentionally

degraded version in which services have been modified and new ones added in order to inject some antipatterns. The changes were performed by a third-party to avoid bias in the results. Given the lack of freely available SBSs, the second round was performed on FraSCAti [5], the runtime support of *Home Automation*. FraSCAti is also an SCA-based system made of more than 90 components and over 130 services scattered between components. A component exposes at least one service and services expose methods. Unlike Home Automation, FraSCAti does not have predefined scenarios–in reality it provides some unit tests, but not complete feature coverage. The detection was performed by instrumenting FraSCAti to produce execution traces as described in Section III-C. As FraSCAti is a runtime support for SOA systems, we loaded and ran diverse SBSs of different technologies (SCA, REST, Web Services, RMI-based) and then, handle these systems to have a maximum feature coverage. The detection of SOA antipatterns in FraSCAti has been performed at the component level instead of service-level since the system architecture is documented at that level while the subsequent validation will be based on this documentation. Moreover, it was empirically established that SCA-based systems suffer from the same architectural flaws as pure SOA systems. Details on the systems including all the scenarios and involved services are available online at http://sofa.uqam.ca/somad.

C. Process

We applied SOMAD for the detection of the six SOA antipatterns on the two targeted SBSs. First, we run the seven scenarios of *Home Automation* on its two versions, and then the six scenarios of FraSCAti. Then, we recreated transactions from the execution traces and run our algorithm for rule generation, with a support of 40% and a confidence of 60%, the corresponding sequential association rules. The step that follows consisted in interpreting the generated association rules. For this purpose, we computed the metrics associated to hypotheses that fit the textual descriptions of the six SOA antipatterns. After this step of interpretation, we obtained for each SBS the list of suspicious services involved in the antipatterns. Finally, we validated the detection results in terms of precision and recall by analyzing the suspicious services manually. Precision estimates the ratio of true antipatterns identified among the detected antipatterns, while recall estimates the ratio of detected antipatterns among the existing antipatterns. This validation has been performed manually by an independent software engineer, whom we provided the descriptions of antipatterns, the two versions of the analyzed system *Home Automation*, and the system FraSCAti with a printed description of its architecture available online on the FraSCAti web site (http://frascati.ow2.org). For both systems, we compared the results with the ones obtained by SODA. For FraSCAti, we reported the detection results to their development team and got their feedback as a objective validation.

D. Results

Table III presents the results for the detection of the six SOA antipatterns on the original and evolved version of *Home Automation*. For each SOA antipattern, the table reports the version analyzed of *Home Automation*, the services detected automatically by SOMAD, the services identified manually by the software engineer, the metric values, the recall and precision, the computation time, and finally, the F-measure [21]. Similarly, Table IV provides the detection results on FraSCAti. We recall that the metric values reported in the tables do not represent absolute values (e.g. for NM, the exact number of methods exposed), but rather elicit what we called the *usage representation* of a SBS. And in particular, the metric values are weighted by the fraction $\frac{support}{confidence}$ for highlighting most confident and supported association rules. Thus, a number of methods (NM) of 2 means that among the generated association rules, there are 2 methods that appear in the rules with a high support and confidence.

E. Details of the Results

We present the detection results of SOMAD while comparing them to SODA, both on the system *Home Automation*. The results with SOMAD are quite similar to the ones obtained with SODA, except for *The Knot* and *Bottleneck Service* antipatterns.

For example, `IMediator` has been detected and identified as a *Multi Service*, both in SOMAD and SODA, because of its high number of methods (NM $\geqslant$ 2), its high number of matches (NMA $\geqslant$ 3.8) and its low cohesion (COH $\leqslant$ 0.5). These metric values have been evaluated as high and low in comparison with the metric values of *Home Automation*. For example, for the metric NM, the boxplot estimates the high value of NM in *Home Automation* as equal to 2. Similarly, the detected *Tiny Service* has a very low number of methods (NM $\leqslant$ 1) and a high outgoing coupling (OC $\geqslant$ 4) according to the boxplot. In the original version of *Home Automation*, we did not detect any *Tiny Service*. An independent engineer extracted one method from `IMediator` and moved it into a new service named `MediatorDelegate`; this newly injected service has been detected as a *Tiny Service*. Two occurrences of *Chatty Service* have been discovered in *Home Automation*, both in SOMAD and SODA. `PatientDAO` and `IMediator` have a high number of matches (NMA $\geqslant$ 3.8), which mean that the service *talks* too much, and they have a high number of different partners (NDP $\geqslant$ 0.6).

`PatientDAO` has been detected as a *Knot* because it has a high cyclic invocation dependencies (CID $\geqslant$ 2) and a low cohesion (COH $\leqslant$ 0.5). The metric CID allows the identification of cyclic invocation dependency. In *Home Automation*, the set of services `PatientDAO1`, `PatientDAO2`, `PatientDAO3`, `PatientDAO4` are tightly coupled because each of them represents a part of a patient's information (name, address, phone number). Therefore, cyclic invocations between these services appear when information about a patient are requested. SOMAD does not report the false positive, `IMediator`, reported by SODA, and thus obtains a better precision for this antipattern.

Three services have been detected as *BottleNeck Services*: `IMediator` `PatientDAO`, and `SunSpotService` be-

TABLE III: Results comparison between SODA & SOMAD on HomeAutomation

Antipattern Name	Automatically detected services		Manually identified services	SOMAD Metrics	Recall	Precision	Time	F_1
Tiny Service	SODA	Mediator-Delegate	Mediator -Delegate	OC $\geqslant$ 4	[1/1] 100%	[1/1] 100%	0.194s	100%
Detected on the Evolved Version	SOMAD	Mediator-Delegate		NM $\leqslant$1	[1/1] 100%	[1/1] 100%	0.077s	100%
Multi Service	SODA	IMediator	IMediator	NM $\geqslant$ 2 NMA $\geqslant$ 3.8	[1/1] 100%	[1/1] 100%	0.462s	100%
	SOMAD	IMediator		COH $\leqslant$ 0.5	[1/1] 100%	[1/1] 100%	0.050s	100%
Chatty Service	SODA	PatientDAO IMediator	PatientDAO	NMA $\geqslant$ 3.8	[2/2] 100%	[2/2] 100%	0.383s	100%
	SOMAD	PatientDAO IMediator	IMediator	NDP $\geqslant$ 0.6	[2/2] 100%	[2/2] 100%	0.077s	100%
The Knot	SODA	PatientDAO IMediator	PatientDAO	CID $\geqslant$ 2	[1/1] 100%	[1/2] 50%	0.412s	66.6%
	SOMAD	PatientDAO		COH $\leqslant$ 0.5	[1/1] 100%	[1/1] 100%	0.077s	100%
BottleNeck	SODA	IMediator PatientDAO	IMediator	IC $\geqslant$ 4	[2/2] 100%	[2/2] 100%	0.246s	100%
	SOMAD	IMediator PatientDAO SunSpotService	PatientDAO	OC $\geqslant$ 3	[2/2] 100%	[2/3] 66%	0.076s	79.5%
Chain Service	SODA	{IMediator, PatientDAO, SunSpotService, PatientDAO2}	{IMediator, PatientDAO, PatientDAO2}	LC $\geqslant$ 4	[3/3] 100%	[3/4] 75%	0.229s	85.7%
	SOMAD	{IMediator, PatientDAO, SunSpotService, PatientDAO2}			[3/3] 100%	[3/4] 75%	0.056s	85.7%
Average	SODA				100%	87.5%	0.231s	92.0%
	SOMAD				100%	90.1%	0.068s	94.2%

TABLE IV: Results comparison between SODA & SOMAD on FraSCAti

Antipattern Name	Automatically detected services		Manually identified services	SOMAD Metrics	Recall	Precision	Time	F_1
Tiny Service	SODA	SCA-Parser	SCA-Parser	OC $\geqslant$ 3	[1/1] 100%	[1/1] 100%	0.083s	100%
	SOMAD	SCA-Parser		NM $\leqslant$1	[1/1] 100%	[1/1] 100%	0.066	100%
Multi Service	SODA	juliac Explorer-GUI	Explorer-GUI	NDP $\geqslant$ 24 NMA $\geqslant$ 70	[1/1] 100%	[1/2] 50%	0.462s	66.67%
	SOMAD	Explorer-GUI		COH $\leqslant$ 0.5	[1/1] 100%	[1/1] 100%	0.050s	100%
Chatty Service	SODA	*not present*	*not present*	NMA $\geqslant$ 70	[0/0] 100%	[0/0] 100%	0.97s	100%
	SOMAD	*not present*		NDP $\geqslant$24	[0/0] 100%	[0/0] 100%	0.77s	100%
The Knot	SODA	SCA-Parser SCA-Composite	SCA-Parser	CID $\geqslant$ 25	[1/1] 100%	[1/2] 50%	1.041s	66.6%
	SOMAD	SCA-Parser		COH $\leqslant$ 0.2	[1/1] 100%	[1/1] 100%	0.7s	100%
BottleNeck	SODA	SCA-Composite SCA-Parser	SCA-Parser	IC $\geqslant$ 3	[2/2] 100%	[2/2] 100%	0.246s	100%
	SOMAD	SCA-Parser SCA-Composite Metamodel-Provider	SCA-Composite	OC $\geqslant$ 3	[2/2] 100%	[2/3] 66.67%	0.076s	80%
Chain Service	SODA	SCA-Parser Composite-Manager Processor	Composite Parser Composite-Manager	LC $\geqslant$ 5	[2/2] 100%	[2/3] 66.67%	0.758	80%
	SOMAD	Composite-Parser Composite-Manager			[2/2] 100%	[2/2] 100%	0.056s	100%
Average	SODA				100%	77.77%	0.707s	85.55%
	SOMAD				100%	94.44%	0.28s	96.6%

cause of their high outgoing and incoming coupling (IC $\geqslant$ 4 and OC $\geqslant$ 3). This time, it is SOMAD that reports the false positive, `SunSpotService`, and thus decreases its precision compared to SODA.

Finally, we detected both in SOMAD and SODA, the transitive chain of invocations `IMediator` $\rightarrow$ `PatientDAO` $\rightarrow$ `PatientDAO2` $\rightarrow$ `SunSpotService` (LC $\geqslant$ 4). In both approaches, the false positive `SunSpotService` has been reported.

We now present the detection results of SOMAD on FraSCAti.

`SCA-Parser` is suspected to be a *Tiny Service* because it includes a low number of methods (NM equal 1) and a high outgoing coupling (OC equal 3). A manual code inspection of FraSCAti revealed that `SCA-Parser` contains only one interface method, named `parse(...)`. The development team of FraSCAti validated this antipattern. They indicated that this service can be invoked alone when only a reading of a SCA file is requested. However, FraSCAti performs more tasks that just reading an SCA file, and these other tasks are performed by other services such as `AssemblyFactory`. This explains the high outgoing coupling.

SOMAD did not detect any *Multi Service* in FraSCAti. However, the manual inspection of FraSCAti allowed the identification of the component `Explorer-GUI` as a *Multi Service*. The FraSCAti development team confirmed that this component uses a high number of services provided by other components of FraSCAti. Indeed, this component encapsulates the graphical interface of FraSCAti Explorer, which aims to provide an exhaustive interface of FraSCAti functionalities. SOMAD was not able to detect it because the execution scenarios did not involve the graphical interface of FraSCAti Explorer.

SOMAD did not detect any *Chatty Service* in FraSCAti. No service has a very high number of matches (NMA) and a very high number of different partners (NDP), respectively higher than 70 and 24. This means that no service appears more than 70 times in the set of association rules and communicates with more than 24 different other services. The manual code inspection confirmed also that there was no *Chatty Service* in FraSCAti. The component `Metamodel-provider` is suspected to be part of a Knot because of its low cohesion (COH $\leqslant$ 0.2) and its very high cyclic invocation dependencies (CID $\geqslant$ 25). The validation by the FraSCAti team has only confirmed that this component was implemented by many other components, but they did not agree on the specification of this antipattern. However, the independent software engineer validated this detection.

SOMAD detected three occurrences of the *BottleNeck Service* antipattern, `SCA-Parser`, `Composite-Parser`, and `Metamodel-provider`, the last of which was identified as a false positive. These services have been identified as *BottleNeck Services* because they have a high outgoing and incoming coupling (OC and IC $\geqslant$ 3). The FraSCAti development team confirmed that `SCA-Parser` is highly used by other services.

Finally, `Composite-Parser` has been detected and identified as a *Chain Service*, whereas `Composite-Manager` is a false positive. `Composite-Parser` is involved in a long transitive chain of invocations (LC $\geqslant$ 4). The FraSCAti development team validated this antipattern and indicated that this service uses a delegation chain to perform its behavior.

We can observe that `Composite-Parser` and `SCA-Parser` are very suspicious services. They are both involved in two antipatterns. These services are highly coupled with other services and are part of a long transitive invocation chain. 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 and high transitivity.

Finally, for both systems, the average computational time of SOMAD is 174ms, whereas the one of SODA is 469ms. SOMAD clearly outperforms SODA. This is explained by the fact that for each service, SODA unstacks and executes a pile of aspects including the code for the computation of metrics whereas SOMAD computes metrics directly on traces using association rules. In conclusion, FraSCAti is performing reasonably well towards the antipattern detection. Few services have been detected as antipatterns compared to the high number of FraSCAti components/services. Mainly, `SCA-Parser` is on the critical path of all processing performed by FraSCAti.

F. Threats to validity

The main threat to the validity of our results corresponds to the *external validity*, i.e., the possibility to generalize the current results to other SBSs. Given the lack of freely available systems, we have done our best to obtain real systems such as FraSCAti and we experimented with two versions of *Home Automation*. However, we plan to run these experiments on other SBSs in the future, with special focus on SBSs implementing other SOA technologies, such as REST and Web services. Regarding the *internal validity*, the detection results depend on our hypotheses. Although we did not perform our experiments on a representative set of antipatterns as done with SODA, we obtained comparable results in terms of precision and recall. The subjective nature of interpreting the association rules and validating antipatterns is a threat to the *construct validity*. We control this threat by specifying our hypotheses based on a literature review on antipatterns and by involving in our study an independent engineer and the FraSCAti development team. Finally, we minimize the threats to *reliability validity* by automating the generation of association rules.

V. Conclusions and Future Work

The detection of SOA antipatterns is a crucial activity if we are to ensure the architectural and overall quality of SBSs. In this paper, we present a new and innovative approach, SOMAD, for the detection of antipatterns. The approach relies on two complementary techniques, from two thriving fields in software engineering, mining system traces and software measurement, respectively, both put in an SOA environment. More precisely, SOMAD detects SOA antipatterns by first discovering strong associations between services from execution

traces and then filtering the resulting knowledge by means of domain-specific metrics. The usefulness of SOMAD was demonstrated by applying it to two independently developed SBS. The results of our approach, were compared to those of its forebear, SODA: The outcome shows that SOMAD is a relevant approach as it is substantially more precise (by a margin ranging from 2.6% to 16.67%) and efficient (2.5+ times faster) while keeping the recall to 100%. Moreover, SOMAD has a wider coverage than SODA as it can adapt to execution traces from any SOA technology –and is potentially applicable to traces produced by OO systems– as opposed to a narrow focus on SCA SBSs.

As a next step, we envision the application of SOMAD in the context of a large data center whereby the goal would be to optimize the data center communications. In the near future, we shall also investigate alternative mining techniques to refine our approach with additional information, e.g. directly extracting architectural overviews with graph pattern mining [35] or, detecting recurring patterns of anomalous behavior with rare pattern mining [36]. Finally, combining explicit semantic representations of SOA antipatterns, e.g. in OWL ontologies, with powerful mining methods for heterogeneous labeled graphs (see [37]) seems to be a particularly promising track for the extraction of complex structural and/or behavioral antipatterns.

ACKNOWLEDGMENT

The authors thank Phillipe Merle and Lionel Seinturier for their help in understanding FraSCAti. This work is partly supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2006.

[2] N. Moha, F. Palma, M. Nayrolles *et al.*, "Specification and detection of soa antipatterns," in *ICSOC*, 2012.

[3] P. Wolfgang, *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.

[4] M. Nayrolles, F. Palma, and Moha, "Soda : A tool for automatic detection of soa antipatterns," in *ICSOC - Tool Paper*, 2012.

[5] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *SPE*, vol. 42, no. 5, pp. 559–583, 2012.

[6] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance From Perfection is a Better Criterion Than Closeness To Evil When Identifying Risky Code," in *Proceedings of the IEEE/ACM ASE*. ACM, 2010, pp. 113–122.

[7] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[8] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE TSE*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

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

[10] M. J. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] D. L. Settas, G. Meditskos, I. G. Stamelos, and N. Bassiliades, "SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies," *ESA*, vol. 38, no. 6, pp. 7633–7646, June 2011.

[12] S. Wong, M. Aaron, J. Segall, K. Lynch, and S. Mancoridis, "Reverse Engineering Utility Functions Using Genetic Programming to Detect Anomalous Behavior in Software," in *Proceedings of WCRE*. IEEE Computer Society, 2010, pp. 141–149.

[13] T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," *JOT*, vol. 7, no. 3, pp. 55–90, April 2008.

[14] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *Proceedings of the 11th WCRE 2004*. IEEE Computer Society, 2004, pp. 112–121.

[15] A. Khan, A. Lodhi, V. Köppen, G. Kassem, and G. Saake, "Applying process mining in soa environments," in *Proceedings of the 2009 ICSOC*. Springer-Verlag, 2009, pp. 293–302.

[16] M. J. Asbagh and H. Abolhassani, "Web service usage mining: mining for executable sequences," in *Proceedings of WSEAS ICACSE - Volume 7*, ser. ACS'07. Stevens Point, Wisconsin, USA: WSEAS, 2007, pp. 266–271.

[17] S. Dustdar and R. Gombotz, "Discovering web service workflows using web services interaction mining," *IJBPIM*, vol. 1, pp. 256–266(11), 27 February 2007.

[18] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," *ICPC*, vol. 0, pp. 84–88, 2006.

[19] A. Yousefi and K. Sartipi, "Identifying distributed features in soa by mining dynamic call trees." in *ICSM*. IEEE, 2011, pp. 73–82.

[20] B. Upadhyaya, R. Tang, and Y. Zou, "An approach for mining service composition patterns from execution logs." *Journal of Software : Evolution and Process*, 2012.

[21] J. K.-Y. Ng, Y.-G. Guéhéneuc, and G. Antoniol, "Identification of behavioural and creational design motifs through dynamic analysis," *JSMRP*, vol. 22, no. 8, pp. 597–627, 2010.

[22] L. Hu and K. Sartipi, "Dynamic analysis and design pattern detection in java programs," in *Proceedings of SEKE'2008*, 2008, pp. 842–846.

[23] G. Piatetsky-Shapiro, "Discovery, analysis, and presentation of strong rules," *KDD*, pp. 229–238, 1991.

[24] W. Stevens, G. Myers, and L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

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

[26] A. Rotem-Gal-Oz, E. Bruno, and U. Dahan, *SOA Patterns*. Manning Publications Co., 2012.

[27] M. Perepletchikov, C. Ryan, and Z. Tari, "The Impact of Service Cohesion on the Analyzability of Service-Oriented Software," *IEEE Transactions on Services Computing*, vol. 3, no. 2, pp. 89–103, Apr. 2010.

[28] J. Pei, J. Han, B. Mortazavi-Asl, Wang *et al.*, "Mining sequential patterns by pattern-growth: The prefixspan approach," *Transactions On Knowledge And Data Enginnering*, vol. 16, no. 11, pp. 1424–1440, 2004.

[29] H. Mannila, H. Toivonen, and A. Inkeri Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, pp. 259–289, 1997.

[30] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu, "Mining access patterns efficiently from web logs," *Knowledge Discovery and Data Mining. Current Issues and New Applications*, pp. 396–407, 2000.

[31] P. Fournier-Viger, R. Nkambou, and V. Tseng, "Rulegrowth: mining sequential rules common to several sequences by pattern-growth," in *Proceedings of the 2011 SAC*. ACM, 2011, pp. 956–961.

[32] P.-m. Fournier and M. R. Dagenais, "Combined Tracing of the Kernel and Applications with LTTng," in *Linux Symposium*, 2009.

[33] N. Wilde, S. Simmons, M. Pressel, and J. Vandeville, "Understanding features in SOA," in *Proceedings of SDSOA '08*. New York, New York, USA: ACM Press, May 2008, p. 59.

[34] A. Yousefi and K. Sartipi, "Identifying distributed features in SOA by mining dynamic call trees," in *ICSM*, Sep. 2011, pp. 73–82.

[35] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 2, 2006.

[36] L. Szathmary, A. Napoli, and P. Valtchev, "Towards rare itemset mining," in *Proc. of the 19th IEEE Intl. ICTAI'07*. IEEE Computer Society, 2007, pp. 305–312.

[37] M. Adda, P. Valtchev, R. Missaoui, and C. Djeraba, "A framework for mining meaningful usage patterns within a semantically enhanced web portal," in *Proc. of C3S2E '10*. ACM, 2010, pp. 138–147.