UNIVERSITÉ DU QUÉBEC À MONTRÉAL

AMÉLIORATION DE LA DÉTECTION D'ANTI-PATRONS DANS LES SYSTÈMES À BASE DE SERVICES PAR LA FOUILLE DE TRACES D'EXÉCUTION

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MATHIEU NAYROLLES

MARS 2014

REMERCIEMENTS

Au cours de ces mois au Canada, de nombreuses personnes m'ont aidé et soutenu afin que je puisse m'intégrer au mieux dans ce nouveau pays ainsi que dans cette nouvelle université. Ce sont ces personnes que je souhaite remercier tout particulièrement.

Pr.Naouel Moha, ma directrice de recherche pour m'avoir initié à la qualité logicielle et aux applications à base de services. Je la remercie également pour sa disponibilité constante et pour m'avoir fait découvrir le monde de la recherche. Pr.Petko Valtchev, mon codirecteur de recherche pour avoir accepté de me coencadrer dans mes recherches et m'avoir initié au monde de la fouille de données.

Pr. Privat, Pr. Beaudry et Pr. Tremblay pour les connaissances acquises dans leurs cours respectifs de principes avancés de langages à objets, planification eb intelligence artificielle et parallélisme hautes performances. Je remercie aussi Messieurs Berger et Levesque d'avoir fait appel à moi pour les assister dans leurs cours respectifs de Programmation Agile et Modélisation avancée.

Alexandre Sbriglio, pilote du cycle supérieur eXia.Cesi d'Aix-En-Provence, pour m'avoir suivi de près malgré la distance et pour m'avoir déchargé de nombreux soucis liés à ma vie au Canada ou aux interactions entre l'UQAM et l'eXia.Cesi.

Pr. Yann-Gaël Guéhéneuc de l'École Polytechnique de Montréal pour m'avoir donné accès aux différentes recherches menées dans son laboratoire, m'avoir invité à certains séminaires, et pour nos discussions sur la généricité en Java. Pr. Abdelwahab Hamou-Lhadj de l'Université de Concordia, pour avoir participé à la

maturation de nos idées et approches basées sur les traces d'exécutions.

Je souhaite aussi remercier Phillipe Merle et Lionel Seinturier pour leurs validations de notre approche sur le système *FraSCAti* et Francis Palma pour la sa validation du système *Home-Automation*.

Les stagiaires du LATECE, étudiants à la maîtrise et au doctorat, qui ont contribué à créer une atmosphère de travail agréable et qui ont toujours été disponibles en cas de besoin. Guillaume, Benjamin, Nicolas, Choukri, Gino, Anis, Anthony, Francis et tous ceux que je ne peux pas citer. Les personnes qui ont contribués à rendre ma vie à Montréal plus qu'agréable : Samuel, Clarisse, David, Tiphaine, Sébastien, Benjamin et tant d'autres.

Caroline Lafarie-Gremillet et Marie De Abreu pour les relectures profondes et complètes de ce mémoire.

Finalement, Lakmé Gremillet pour m'avoir soutenu au travers de cette maîtrise et ce, quel que soit les doutes et les difficultés.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Les systèmes à base de services (SBSs), à l'instar des autres systèmes complexes, évoluent pour s'adapter à des nouvelles demandes utilisateurs et à de nouveaux contextes d'exécution. Cette évolution continue peut facilement détériorer la qualité de service (QoS) et de conception des SBSs et introduire des défauts de conception, connus sous le nom d'anti-patrons SOA (*Service Oriented Architecture*). Les anti-patrons de conception sont des solutions récurrentes et reconnues sous-optimums à des problèmes connus. Les anti-patrons sont donc l'inverse des patrons de conception qui sont de bonnes solutions à des problèmes connus. Les anti-patrons SOA conduisent à une maintenabilité et une réutilisabilité réduites des SBSs. Il est donc important de les détecter puis de les supprimer. Cependant, les techniques pour leur détection en sont à leurs balbutiements, et il n'y a actuellement qu'un seul outil, nommé SODA (*Service Oriented Detection for Anti-patterns*), permettant leur détecion automatique. SODA est basé sur un ensemble de métriques majoritairement statiques et sur quelques métriques dynamiques qui sont calculées lors de l'exécution du système cible grâce à des techniques de programmation orientée aspect. Dans ce mémoire, nous proposons une nouvelle approche nommée SOMAD (*Service Oriented Mining for Antipatterns Detection*) qui est une évolution de SODA. Le but de notre nouvelle approche est d'améliorer la détection automatique des anti-patrons SOA en fouillant les traces d'exécution que produisent les SBSs. En effet, les traces d'exécution représentent plusieurs avantages qui permettront d'améliorer la détection car elles permettent de capturer pleinement la nature hautement dynamique des SBSs tout en nécessitant un niveau de contrôle relativement faible sur les systèmes cibles. SOMAD mine des règles d'association pertinentes dans les traces d'exécution des SBSs, puis les filtre via une suite de métriques dédiées. Nous discutons d'abord les modèles de règles d'association sous-jacents et les intuitions soutenant les métriques dédiées aux SBSs. Les règles d'association permettent de découvrir des relations entre différents objets dans un grand ensemble de données. Nous présentons aussi deux expérimentations visant la validation formelle de notre approche. Une comparaison entre SOMAD et SODA est effectuée et révèle l'efficacité de SOMAD face à SODA : sa précision est meilleure d'une marge allant de 8.3% à 20% tout en gardant le rappel à 100%. Finalement, SOMAD est, au minimum, 2.5 fois plus rapide que SODA sur les mêmes sujets d'expérimentation.

INTRODUCTION

Contexte de l'étude : Les applications à base de services et les anti-patrons

Les systèmes à base de service (SBSs) sont composés de services déjà prêts et accessibles par Internet (Erl, 2008). Les services sont des unités logicielles autonomes, interopérables, et réutilisables qui peuvent être implémentées en utilisant un large choix de technologies tels que les services web, REST (*REpresentational State Transfert*), ou SCA (*Service Component Architecture*, une surcouche au SOA). La plupart des plateformes web connues du grand public sont de parfaits exemples de SBSs comme Amazon, PayPal et eBay. De tels sytèmes sont complexes — ils génèrent des flôts massifs de communication entre les services — et hautement dynamiques : des services apparaissent, disparaissent ou sont modifiés. L'évolution constante des SOAs (*Service Oriented Architecture Systems*) peut facilement détériorer la qualité de leurs architectures par l'introduction de défauts architecturaux connus sous le nom d'anti-patrons SOA (Moha *et al.*, 2012). Un anti-patron de conception est une solution connue et non-optimale à un problème connu. Les anti-patrons sont opposés aux patrons de conception qui sont eux des bonnes pratiques en réponse à des problèmes connus. Les anti-patrons peuvent donc être qualifiés de mauvaises pratiques de conception.

Par exemple, le *Tiny Service* est un anti-patron largement répandu dans les systèmes à base de services. Ce service a une très petite taille avec très peu de méthodes qui implémentent seulement une partie d'une abstraction (Dudney, 2003).

Les *Tiny Services* sont généralement accompagnés de plusieurs autres services fortement couplés, ce qui induit une complexité dans le développement et réduit la réutilisabilité. De plus, il a été prouvé que les *Tiny Services* sont une des principales raisons d'échecs (*failure*) des systèmes à base de services (Kral et Zemlicka, 2009).

Problème étudié : La détection des anti-patrons

Étant donné l'impact néfaste des anti-patrons sur la réutilisabilité et la maintenabilité des SBS, il existe un besoin clair et urgent de techniques et d'outils visant leur détection. Cependant, la nature hautement dynamique et distribuée des SBSs rend la détection automatique des anti-patrons SOA compliquée. C'est un véritable défi, surtout en comparaison avec d'autres outils visant la détection d'anti-patrons dans les systèmes objets (Marinescu, 2004; Fokaefs *et al.*, 2007; Moha *et al.*, 2010). En 2012, notre équipe, composée de Naouel Moha, Francis Palma, Benjamin Joyen-Conseil, Yann-Gaël Guéhéneuc, Benoit Beaudry, Jean-Marc Jézéquel et moi même, a développé une approche nommée SODA (*Service Oriented Detection for Anti-patterns*) (Nayrolles *et al.*, 2012; Moha *et al.*, 2012) qui vise la détection des anti-patrons SOA. Cette approche repose sur un langage spécifique au domaine (*Domain Specific Language*, DSL) pour spécifier les anti-patrons SOA et est basée sur des métriques (majoritairement statiques) d'un côté, et sur une méthode de génération automatique d'algorithmes de détection, de l'autre.

Bien qu'étant efficace et précise, SODA souffre de sérieuses limitations. En effet, SODA exécute deux phases d'analyse, une première statique, suivie d'une seconde, dynamique. La première analyse statique requiert un accès aux interfaces des services. En conséquence, SODA ne peut pas analyser dont les sources

ne sont pas disponible. La seconde phase, à moindre portée, d'analyse dynamique requiert l'éxécution concrète du système et de ce fait, la création de scénarios exécutables. De plus, SODA a été créé spécifiquement pour les systèmes de type SCA et sa précision tend à faiblir lorsque la taille des systèmes augmente. Étant donné les limitations de SODA, il y a un espace pour l'amélioration de nos approches et outils pour la détection d'anti-patrons SOA. Ces améliorations doivent apporter une détection précise, efficace et applicable à toutes les technologies SOA (REST, SCA, service web, ...) et ce quelque soit la taille du système. Dans ce mémoire, nous proposons une approche nommée SOMAD (*Service Oriented Mining for AntiPatterns Detection*). Cette approche ne requiert pas de scénarios, à l'inverse de SODA, et repose uniquement sur les traces d'exécution qui peuvent être disponibles dans toutes les technologies SOA. SOMAD est capable d'éliminer les données non pertinentes en utilisant une technique de fouille de données : la fouille de règles d'association séquentielle. La fouille de règles d'association (Agrawal et Srikant, 1994) et a fortiori la fouille de règles d'association séquentielles – en particulier l'algorithme *RuleGrowth* (Fournier-Viger *et al.*, 2011) – sont des méthodes servant à découvrir des relations intéressantes dans de larges bases de données. SOMAD applique ses méthodes sur les traces d'exécution pour découvrir des anti-patrons SOA sous forme de configurations particulières dans la composition des règles d'association. Pour ce faire, nous utilisons une variante de la fouille de règles d'association basée sur les séquences ou épisodes. Dans notre cas, les séquences représentent des suites d'appels de services et de méthodes. Ensuite, nous filtrons ces règles d'association en utilisant une suite de métriques dédiées afin d'extraire la connaissance pertinente des règles d'association.

Contributions

Les principales contributions liées à ce mémoire sont les suivantes :

1. Une nouvelle approche nommée SOMAD pour la détection des anti-patrons SOA. Cette approche est basée sur des règles d'association fouillées sur des traces d'exécution qui peuvent provenir de toutes les technologies SOA.
2. une validation empirique de notre approche qui démontre l'amélioration apportée par SOMAD en terme de précision (8.3% à 20%) et de vitesse (2.5 fois plus rapide).
3. Une évolution de l'algorithme *RuleGrowth* visant la fouille de règles d'association séquentielles dans des séquences d'appels des SBSs. De ce fait, nous améliorons la détection d'anti-patrons SOA.

Les contributions 1 et 2 ont été présentées à la 20^{ieme} édition de la conférence internationale de travail sur la rétro-ingénierie (WCRE 2013, *Working Conference on Reverse Engineering*) (Nayrolles *et al.*, 2013).

Structure du document

Ce mémoire est organisé de la manière suivante. Les deux premiers chapitres présentent respectivement l'état de l'art sur la détection de patrons de conceptions et l'extraction de connaissances depuis les traces d'exécution. Le troisiéme chapitre, quant à lui, présente l'approche SOMAD. Le quatriéme présente l'implémentation de SOMAD tandis que le cinquième et dernier chapitre présente nos expérimentations. Enfin, nous proposons quelques remarques de fin dans la conclusion.

CHAPITRE I

ÉTAT DE L'ART SUR LA DÉTECTION DE PATRONS DE CONCEPTION

Conserver une bonne qualité architecturale est essentiel pour construire des systèmes maintenables et évolutifs. Les patrons et anti-patrons ont été reconnus comme une des meilleures façons d'exprimer ces préoccupations architecturales. Cependant, au contraire des anti-patrons orientés objet, la détection de leurs équivalents orientés services en est encore à ses débuts.

Dans ce chapitre dédié à l'état de l'art sur la détection de patrons de conception, nous verrons tout d'abord la détection des anti-patrons objet suivi par la détection de patrons SOA. Ensuite, une troisième sous section couvrira l'état de l'art de la détection d'anti-patrons SOA. Finalement, nous expliquerons le fonctionnement du seul outil, nommé SODA, permettant la détection automatique d'anti-patrons SOA.

1.1 Détection d'anti-patrons orientés objets

Ce champs de recherche est toujours largement ouvert, même si les contributions récentes se révèlent plus incrémentales que réelement nouvelles.

Un nombre important d'approches et d'outils existent pour la détection d'anti-

patrons objet (Lanza et Marinescu, 2006; Moha *et al.*, 2010; Kessentini *et al.*, 2010, 2011) et de nombreux livres ont aussi porté sur le sujet. En effet, Brown *et al.* (1998) ont produit un catalogue de 40 anti-patrons tandis que Beck, dans le livre à succès *Refactoring* de Fowler *et al.* (1999), a identifié 22 mauvaises odeurs de code (ou *code smells* en anglais) qui doivent être traquées et éliminées afin d'avoir un code de meilleure qualité. Un exemple simple d'anti-patron orienté objet est le *"blob"*, aussi connu sous le nom de *"god object"* (figure 1.1), correspond à un contrôleur de grande taille (grand nombre d'attributs et de méthodes) qui dépend de données stockées dans des classes adjacantes.

Le blob est une très grande classe qui déclare de nombreux champs et méthodes avec une faible cohésion. Une classe de type contrôleur monopolise la majorité du traitement effectué par le système, prend la majorité des décisions et dirige le traitement efféctué par les autres classes. De plus, il est fortement couplé aux classes de données adjacentes.

Afin de détecter un blob dans un programme à base d'objets, il faut identifier le nombre de classes de données qui entourent un controleur, calculer sa cohésion, le nombre de champs et de méthodes déclarés.

Au sein des travaux sur les anti-patrons objet, certains sont particulièrement intéressants pour nos objectifs. Notamment DECOR (Moha *et al.*, 2010) qui est une approche basée sur des règles visant la spécification et la détection de motifs dans le code ou la conception des systèmes objets. Les motifs sont des morceaux de code ou de conception qui sont reconnaissables, car ils ont été identifiés et nommés dans le but de faciliter la communication entre les membres d'une même équipe et d'améliorer la qualité logiciel en général. Les auteurs de cette étude utilisent un langage spécifique au domaine (ou DSL) pour spécifier les motifs et, ensuite, ils génèrent automatiquement des algorithmes de détection qui sont di-

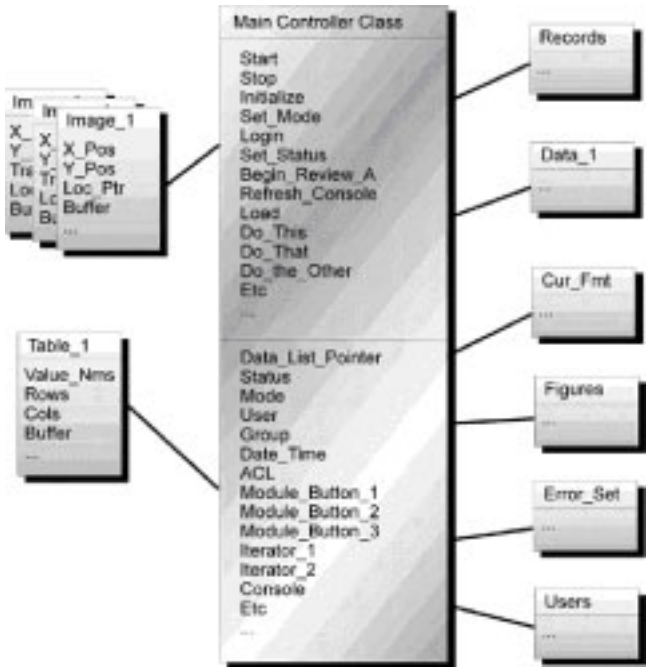


Figure 1.1: Le blob objet

rectement exécutables. DECOR peut détecter les anti-patrons objets avec une précision de 60.5% et un rappel de 100%. Une autre approche proposée par Kessentini *et al.* (2011) a obtenues de meilleurs résultats que DECOR et a apporté une construction automatique des règles de détection. De plus, les auteurs ont utilisé des algorithmes génétiques pour maximiser la détéction via l'optimisation des ensembles de règles. Les algorithmes génétiques imitent le processus de la sélection naturelle pour produire des solutions approchant le résultat optimal en un temps raisonnable. Finalement, Khomh *et al.* (2011) ont à nouveau obtenu de meilleurs résultats en utilisant des réseaux bayésiens. Les réseaux bayésiens sont un modèle probabiliste qui représentent des variables aléatoires et leurs dépendances conditionnelles grâce à un graphe dirigé acyclique (DAG). Une alternative à la spécification par un langage dédié, nommée SPARSE, a été présentée par Settas *et al.* (2011). SPARSE permet de décrire les anti-patrons en utilisant des onto-

logies OWL agrémentées avec des règles SWRL (*Semantic Web Rule Language*) tandis que leurs occurrences sont testées en utilisant le raisonneur sémantique Pellet (Sirin *et al.*, 2007). Un raisonneur sémantique est capable de déduire des conséquences logiques depuis un ensemble de faits avérés.

D'autres travaux pertinents se sont focalisés sur la détection d'anti-patrons spécifiquement liés aux performances et aux ressources systèmes. Par exemple, (Wong *et al.*, 2010), utilisent un algorithme génétique pour la détection de défauts dans les logiciels. Dans un autre travail pertinent, Parsons (2007) s'occupe de la détection d'anti-patrons de performance. Il utilise une approche basée sur des règles statiques et dynamiques visant les applications à base de composants (plus particulièrement les applications Java EE [1]).

De plus, il existe une grande variété d'outils développés par l'industrie et la communauté académique qui visent la détection automatique d'anti-patrons dans les systèmes objet ; les plus connus étant : FindBugs, iPlasma, JDeodorant, PMD et SonarQube (Rutar *et al.*, 2004).

1.2 Détection de patrons SOA

Le catalogue actuel de patrons SOA est relativement riche. En effet, il existe de nombreux livres (Erl, 2009; Daigneau, 2011) portant sur ce sujet et plus encore (Rotem-Gal-Oz *et al.*, 2012). Ces ouvrages fournissent de bonnes pratiques à adopter pour concevoir des applications à base de services. Par exemple, Rotem-Gal-Oz *et al.* (2012) introduisent 23 patrons SOA et quatre anti-patrons suivi de discussions sur les raisons de leurs apparitions et les solutions & problèmes qu'ils

1. Java Enterprise Edition, ou Java EE (anciennement J2EE), est une spécification pour la technologie Java de Sun Microsystems (Oracle) plus particulièrement destinée aux applications d'entreprise.

peuvent apporter. Erl, quant à lui, introduit plus de 80 patrons SOA séparés en quatre catégories : architecturaux, implémentation, sécurité et gouvernance (Erl, 2009). Malgré ce catalogue de patrons relativement dense, peu de techniques ont été proposées pour la détection de patrons dans un environnement SOA.

Deux contributions sont particulièrement pertinentes dans le domaine de la détection de patrons SOA. Tout d'abord Upadhyaya *et al.* (2012) ont identifié des patrons de composition de services, c'est à dire, des services qui sont utilisés ensemble de façon répétée tout en étant structurellement et fonctionnellement similaires.

Ces travaux pourraient également être adaptés pour la correction d'anti-patrons. La seconde approche est nommée SODOP (*Service Oriented Dection Of Patterns*) et a été proposée par Demange *et al.* (2013). Cette approche, basée sur l'approche SODA (Moha *et al.*, 2012; Nayrolles *et al.*, 2012; Palma, 2013), propose de détecter cinq patrons nouvellement définis en utilisant des cartes de règles. Les cartes de règles sont des ensembles de métriques qui peuvent être statiques ou dynamiques, comme le couplage ou la cohésion. Par la suite, ils génèrent des algorithmes de détection et les appliquent sur un système à base de services instrumentalisé pour exécuter des scénarios. Une explication approfondie de SODA, l'approche sur laquelle SODOP est basée, sera fournie dans la section 1.4. Outre ces deux contributions majeures, il existe quelques travaux sur la détection de patrons SOA en utilisant la similarité entre les services (Liang *et al.*, 2006), ou les *workflows* (Weijters et van der Aalst, 2003; Dijkman *et al.*, 2009). Cependant, ces travaux sont différents, car ils ne cherchent pas à évaluer la qualité globale d'un système, au contraire de SODOP. En effet, SODOP est la seule approche qui vise clairement à déterminer la qualité de conception de systèmes à base de services dans le but de faciliter la maintenance et l'évolution de tels systèmes.

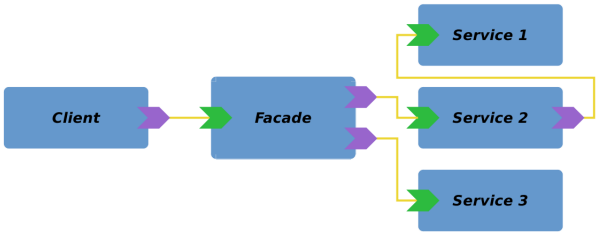


Figure 1.2: Le patron facade (Demange *et al.*, 2013).

A titre d'exemple, le façade (figure 1.2) est un patron SOA permettant d'obtenir une abstraction supérieure. Ce patron est inspiré par des patrons similaires nommés Remote Facade (Fowler *et al.*, 1999) et Decoupled Contract (Erl, 2009). Une facade peut être responsable de l'orchestration du système ou être utilisée pour masquer des systèmes légataires.

Pour détecter un patron façade, une des méthodes consiste à calculer son temps de réponse et son ratio de couplage entrant sur le couplage sortant. Une façade devrait avoir un fort temps de réponse puisqu'elle offre un point d'entrée au système à de nombreux clients et un faible ratio couplage entrant / sortant puisqu'elle cache l'implémentation de nombreux autres services (Demange *et al.*, 2013).

1.3 Détection d'anti-patrons SOA

Contrairement aux patrons SOA, les anti-patrons ont beaucoup moins été étudiés par la communauté. En effet, la littérature sur le sujet est plutôt réduite. De ce fait, la plupart des références sont des pages Internet où des développeurs SOA partagent leurs expériences sur les bonnes pratiques SOA (Cherbakov *et al.*, 2005). Il existe tout de même quelques ouvrages conséquents, notamment (Dudney, 2003) qui constitue le premier livre sur les anti-patrons SOA. Cet ouvrage répertorie

53 anti-patrons liés à l'architecture et l'implémentation de systèmes JEE (*Java 2 Platform Enterprise Edition*), tels que les EJB (*Enterprise Java Beans*, JSP (*JavaServer Pages*) ou encore les *Servlets*. Malgré la pertinence de ce livre pour nos recherches, il ne propose aucune approche pour la détection automatique de ces patrons. De plus, ces anti-patrons sont uniquement applicables aux systèmes JEE alors que la plupart d'entre eux ne sont que des variantes des *Tiny Service* et *Multi Service* évoqués dans l'introduction. Kral et Zemkicka ont eux aussi apporté une contribution significative aux anti-patrons SOA. En effet, ils ont spécifié sept anti-patrons SOA directement liés à l'utilisation de pratiques objets. Une fois encore, la question de la détection automatique de ces anti-patrons n'est pas évoquée (Kral et Zemlicka, 2009).

Bien que le catalogue global d'anti-patrons SOA commence à gagner un certain intérêt et grossit de jour en jour, seulement de quelques contributions existent pour la détection automatique d'anti-patrons dans des environnements SOA. En effet, seulement deux contributions sont à signaler (Trčka *et al.*, 2009) en plus de la nôtre — SODA — décrite dans trois publications différentes (Moha *et al.*, 2012; Nayrolles *et al.*, 2012; Palma *et al.*, 2013). D'autres auteurs proposent une technique pour découvrir des anti-patrons dépendant des flux de données, comme par exemple, des données manquantes, inconsistantes ou encore supprimées trop tôt. Ils détectent ces anti-patrons spécifiques en analysant les dépendances entre les données dans les flux d'exécution et les problèmes qui pourraient apparaître en cas de manipulation non optimale (Trčka *et al.*, 2009). Cette étude, bien que pertinente pour nos recherches, se concentre sur les anti-patrons SOA liés aux flux de données alors que nous cherchons à détecter les anti-patrons SOA distribués dans tout le système et non relié à une activité particulière.

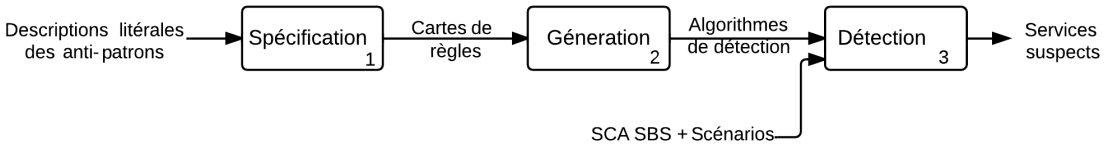


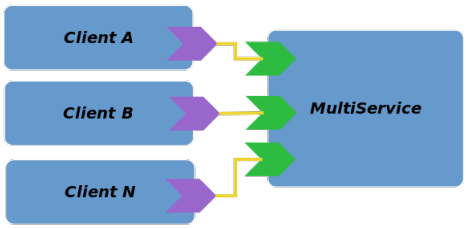
Figure 1.3: L'approche SODA

1.4 SODA, l'approche de l'état de l'art pour la détection automatique d'anti-patrons SOA.

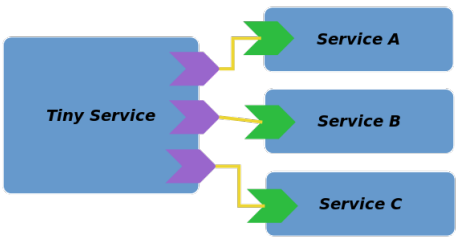
Comme mentionné plus haut, l'unique approche automatique disponible pour la détection d'anti-patrons SOA est SODA (Moha *et al.*, 2012; Nayrolles *et al.*, 2012; Palma *et al.*, 2013). SODA repose sur un langage de règles qui permet la spécification d'anti-patrons en utilisant un ensemble de métriques. Un processus générique transforme les spécifications en algorithmes de détection à exécuter sur les systèmes à analyser. SODA est composé des trois étapes décrite ci-aprés et illustrées par la figure 1.3 :

Spécification d'anti-patrons SOA : Cette étape identifie les propriétés relevant de la spécification des anti-patrons SOA. Ces descriptions textuelles sont présentées dans le tableau 1.1. Les propriétés correspondent de manière générale à des métriques, par exemple, la cohésion, le couplage, le nombre de méthodes, le temps de réponse et la disponibilité. De plus, ces propriétés sont utilisées comme base d'un DSL, qui prend la forme d'un langage à base de règles pour la spécification d'anti-patrons SOA. La spécification finale est une *carte de règles*, qui est une composition de règles combinant des métriques.

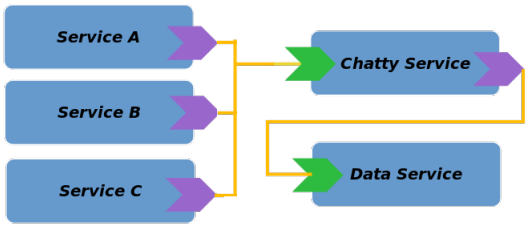
Multi-Service, aussi connu sous le nom “God object” dans le paradigme objets, correspond à un service qui implémente une **multitude de méthodes** faisant référence à différentes abstractions métiers et techniques. Il agrège beaucoup d'abstractions différentes à l'intérieur d'un un même service. Un tel service est difficilement réutilisable à cause de la **faible cohésion** entre ses méthodes et il est souvent non disponible aux utilisateurs finaux à cause de sa charge, laquelle peut introduire de fort temps de réponse (Dudney, 2003).



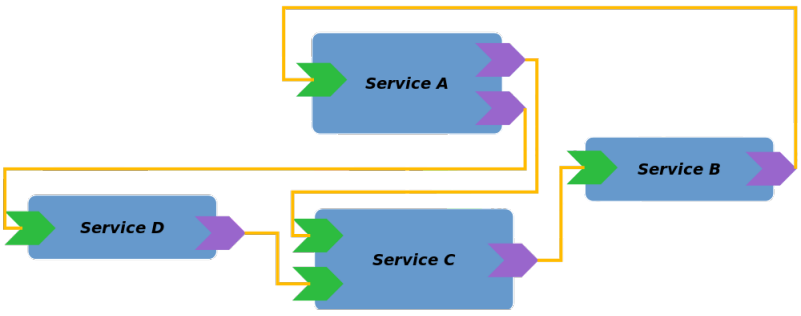
Le **Tiny Service** est un petit service avec **peu de méthodes** qui implémentent uniquement une partie d'une abstraction. Un tel service requiert plusieurs services couplés pour être utilisé correctement. De ce fait, le Tiny Service introduit une complexité importante dans le développement. Dans certains cas extrême, le Tiny Service est limité à **une méthode unique**, avec pour conséquence de nombreux services implémentant l'ensemble des fonctionnalités sous-jacentes (Dudney, 2003).



Le **Chatty Service** correspond à un service qui **échange beaucoup de données** de type primitif. Le Chatty Service est aussi caractérisé par un **grand nombre d'invocations** ; il *discute* énormément avec le reste du système (Dudney, 2003).



Le **Knot** est un ensemble de services **très peu cohésifs** et **fortement couplés entre eux**. Ces services sont de ce fait, très difficilement réutilisables. À cause de la complexité induite de l'architecture, la disponibilité de ces services peut être faible et leur temps de réponse élevé (Rotem-Gal-Oz *et al.*, 2012).



Le **Bottleneck Service** est un service **très utilisé** par le reste du système (les autres services ou clients). Il a un **fort couplage entrant et sortant**. Son temps de réponse peut être élevé car il est utilisé par de nombreux clients, et les clients doivent attendre la réponse des couches inférieures. De plus, sa disponibilité peut être faible à cause du trafic engendré.

Le **Service Chain**, aussi connu sous le nom de "Message Chain" dans les systèmes objets, correspond à **une chaîne de services**. Le Service Chain apparaît quand la requête d'un client est complétée par une invocation consécutive et successive de services. Ce genre de **chaînes de dépendances** engendrent des **invocations transitives**.

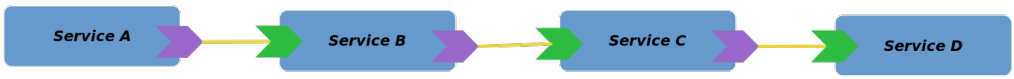


Tableau 1.1: Anti-patrons SOA (Moha *et al.*, 2012) .

Génération des algorithmes de détection : Le but de cette étape est de générer automatiquement des algorithmes de détection en visitant les cartes de règles spécifiées à l'étape précédente. Ce processus est automatique et génère des algorithmes de détection directement exécutables.

Détection des anti-patrons SOA : Cette troisième et dernière étape consiste à appliquer les algorithmes générés sur un système à base de services. Cette étape

permet la détection automatique des anti-patrons SOA en utilisant un ensemble de scénarios prédéfinis qui invoquent les interfaces des services. À la fin de cette étape, les services du système suspectés d'être impliqués dans un anti-patron SOA sont identifiés.

Bien qu'efficace et précis, SODA est une approche intrusive, car elle requiert un ensemble de scénarios qui invoquent concrètement les interfaces des services du système. De plus, son analyse dynamique implique des propriétés propres aux systèmes SCA (*Service Component Architecture*, une sourcouche au SOA). En effet, l'analyse dynamique de SODA repose sur le tissage d'aspects qui est une fonctionnalité de la programmation orientée aspects qui consiste à insérer du code, qui s'exécutera avant ou après une méthode donnée, dans le code métier de l'application visée. SCA est la seule technologie SOA à supporter cette fonctionnalité.

1.5 Conclusion

De nombreux travaux ont été menés pour la détection d'anti-patrons dans les systèmes orientés objets, notamment les approches dérivant ou améliorant celle de (Moha *et al.*, 2010). Pour le monde des applications à base de services, la communauté a créé un catalogue très dense de patrons SOA, mais une seule approche vise leur détection dans le but de déterminer la qualité globale du système pour faciliter la maintenance et l'évolution de tels systèmes. Cette approche présentée par *Demange et al.* a été inspirée par une autre approche nommée SODA – que nous avons proposée en 2012 (Moha *et al.*, 2012; Nayrolles *et al.*, 2012; Palma *et al.*, 2013). SODA se concentre, elle aussi, sur la facilitation de la maintenance et de l'évolution des applications à base de services, mais en détectant les anti-patrons SOA. Malgré que ces deux approches soient pertinentes et efficaces, elle sont limitées par plusieurs points :

— Elles nécessitent d'avoir un bon niveau de contrôle sur le système analysé ; en particulier elles doivent avoir accès aux interfaces des services du système ;
— Elles nécessitent de posséder des scénarios pertinents ;
— SODA est majoritairement basé sur des métriques statiques alors que SODOP utilise une part plus importante de métriques dynamiques ;
— Techno-centrique (SCA) puisque basé sur le tissage d'aspects.

Notre objectif est de reprendre les forces de ces approches comme la spécification déclarative des anti-patrons via un language dédié et la génération automatique d'algorithmes de détection. Néanmoins, nous voulons nous abstraire de la technologie du système analysé, en nous appuyant sur des données qui reflètent la nature hautement dynamique des SBSs. Nous souhaitons aussi éviter toute intrusion dans le fil d'exécution normal du système visé.

CHAPITRE II

ÉTAT DE L'ART SUR L'EXTRACTION DE CONNAISSANCES À PARTIR DE TRACES D'EXÉCUTION

Les traces d'exécution sont à la base de la majorité des analyses dynamiques de systèmes complexes. Les traces d'exécution peuvent être collectées grâce à l'exécution normale – en opposition à l'exécution contrôlée ou dirigée – du système analysé. Cependant, les traces d'exécution sont réputées être difficiles à appréhender à cause de la quantité considérable de données qu'elles contiennent. En effet, l'exécution d'un système de taille moyenne peut potentiellement produire des millions de traces, chacune étant une composition d'information telle que l'horodatage (*timestamp* en anglais), l'identification du client ou du sous-système courant. Au fil des années, des techniques du domaine de la fouille de données ont été appliquées aux traces d'exécution afin d'en extraire des informations intérèssantes.

Dans ce chapitre couvrant l'état de l'art de l'extraction de connaissances à partir de traces d'exécution, nous allons, dans un premier temps, faire une revue des travaux portant sur l'extraction de connaissances dans des environnements orientés services. Dans un second temps, nous introduirons les notions de fouille de règles d'association classiques et séquentielles.

2.1 Extraction de connaissances depuis des traces d'exécution

Un nombre important d'études se sont concentrées sur l'extraction des connaissances contenues dans les traces d'exécution. Ces études ont étés motivées par l'identification d'aspects (Tonella et Ceccato, 2004), de processus d'affaires (Khan *et al.*, 2010), de patrons d'utilisation de services (Asbagh et Abolhassani, 2007), et l'identification de fonctionnalités à la fois dans les systèmes orientés objets (Dustdar et Gombotz, 2006) et services (Safyallah et Sartipi, 2006). Tonella et Cecetato (2004) ont identifié les aspects en générant des traces liées à l'exécution des principales fonctionnalités d'un système. Ensuite, ces traces d'exécution sont comparées aux unités du système qui ont été utilisées — ce qui induit un contrôle sur le code source — via l'analyse formelle de concepts. Le treillis résultant permet de détecter les différents aspects de l'application (Tonella et Ceccato, 2004). Khan *et al.* (2010) fouillent les processus d'affaires dans un environnement SOA en identifiant les traces relatives à un processus. Ces traces appelées "traces de processus" sont ensuite soumises à de nombreuses conversions et analyses sémantiques afin d'en retirer les différents processus d'affaires (Khan *et al.*, 2010). Asbagh et Abolhassani (2007) fouillent des patrons d'exécution séquentiels dans les séquences d'utilisation de services dans le but d'en extraire des patrons d'utilisation généraux et proposent un algorithme performant pour cette tâche spécifique.

Une étude particulièrement importante, car proche de notre approche, porte sur l'identification de composition de services (Upadhyaya *et al.*, 2012), c'est-à-dire, des services qui sont utilisés ensemble de façon répétée tout en étant structurellement et fonctionnellement similaires. Les auteurs détectent neuf compositions de services en utilisant l'analyse de traces d'exécution et l'algorithme Apriori (voir section 2.2.1) pour les détecter. Ils identifient aussi les patrons de composition qui sont structurellement ou fonctionnellement proches afin de les représenter à un

niveau d'abstraction supérieur.

Un petit nombre de projets ont exploré la détection de patrons de conception basée sur la fouille de traces. Ng *et al.* (2010) ont proposé MoDeC, une approche pour identifier des patrons de conception comportementaux et de création en utilisant une analyse dynamique et de la programmation par contraintes. Ils ont réalisé une rétro-ingénierie des scénarios d'utilisation en instrumentant le *bytecode*[1] et ont appliqué des techniques de programmation par contraintes pour détecter des patrons de collaboration à l'exécution. Hu et Sartipi (2008) se sont attaqués à la détection de patrons de conception dans les traces d'exécution en utilisant des scénarios, de la fouille de patrons et de l'analyse formelle de concepts. Leur approche est guidée par un ensemble de scénarios uniques par fonctionnalité de l'application afin d'identifier des patrons par fonctionnalité.

Bien que différentes dans leurs buts et étendues, les études présentées ci-dessus portant sur des anti-patrons et patrons orientés objets forment une base d'expertise et de savoir technique pour créer de nouvelles méthodes visant la détection d'anti-patrons SOA. Malgré le nombre important de ressemblances, les techniques de détection pour les anti-patrons objets ne peuvent pas être directement appliquées pour les services. En effet, les systèmes orientés services utilisent les services comme bloc de construction et, de ce fait, se placent à une abstraction supérieure à l'objet. De plus, la nature hautement dynamique et distribuée des systèmes à base de services soulève des défis qui ne sont pas prépondérants dans les systèmes objets. De manière générale, ces défis sont liés à des difficultés à établir l'ordre des événements, tout comme le comportement stochastique dont de tels systèmes sont

---

1. Le *bytecode* est un code intermédiaire, plus concret (plus proche des instructions machines) que le code source, qui n'est pas directement exécutable. Il est contenu dans un fichier binaire qui représente un programme, tout comme un fichier objet produit par un compilateur.

capables. De ce fait, les techniques établies pour le paradigme objet ne peuvent pas être appliquées.

2.2 Introduction à la fouille de règles d'association

Dans le domaine de la fouille de données, la fouille de règles d'association (*Association rule Mining*, ARM) est une méthode reconnue pour découvrir des co-occurrences entre les attributs d'objets dans des bases de données colossales (G. Piatetsky-Shapiro, 1991). Les règles d'association classiques sont représentées par $A \rightarrow B$, où A et B sont des ensembles d'attributs. En d'autre mots, la fouille de règles d'association cherche à extraire — depuis un ensemble de transactions composées d'items — les items qui apparaissent souvent ensemble (itemsets fréquents) et des règles antécédent $\rightarrow$ conséquent prédisant l'occurrence d'un item d'après les occurrences d'autre items dans la transaction (règles d'association).

La force d'une règle est mesurée par une métrique de *confiance*. La confiance mesure la fréquence de B dans les transactions ayant déjà A (Equation 2.1). Dans cette équation $\sigma(A \cup B)$ représente le nombre d'apparitions de A et B dans la même transaction et $\sigma(A)$ le nombre de transaction dans lesquelles A apparaît.

$$\textit{Confiance (A entraine B)} = \frac{\sigma(A \cup B)}{\sigma(A)} \tag{2.1}$$

L'importance de la règle, quant à elle, c'est-à-dire, combien de fois le motif correspondant appararaît dans les traces, est mesurée par le *support* (Equation 2.2). Dans cette seconde équation, la signification de $\sigma(A \cup B)$ est identique et $|T|$ est le nombre de transactions.

$$\textit{Support (A entraine B)} = \frac{\sigma(A \cup B)}{|T|} \tag{2.2}$$

Pour s'assurer qu'uniquement les règles avec un fort potentiel d'information soient retenues, la fouille est encadrée par des seuils minimaux à atteindre pour les deux métriques.

2.2.1 L'algorithme Apriori

L'algorithme Apriori – publié en 1994 – est, probablement, l'algorithme le plus populaire pour extraire de telles règles d'association (Agrawal et Srikant, 1994). Voici un aperçu de son mode opératoire. Pour chaque transaction ($t \in T$) du tableau 2.1, nous prenons en compte chaque article acheté sans préoccupation de quantité.

Transaction	Items
1	Pain, Lait
2	Pain, Couche, Bière, Oeufs
3	Lait, Couche, Bière, Coke
4	Pain, Couche, Lait, Bière
5	Coke, Lait, Pain, Couche

Tableau 2.1: Table de transactions.

Ensuite, l'algorithme génère les ensembles d'items par taille et ce en fonction d'un support minimum. Ici le support $supp(x) = \frac{\sigma(x)}{|T|}$ représente la fréquence d'apparition de l'itemset (it) dans l'ensemble des transactions (T). Le résultat de cette opération est présenté par le tableau 2.2. Par exemple, l'article {Pain} est présent dans quatre des cinq transactions. Il a donc un support de 80%. Les itemssets fréquents peuvent aussi être composés de plusieurs articles ; par exemple "Bière" et "Couche" apparaissent dans trois des cinq transactions donc {Bière, Couche} a un support de 60%.

A partir de ces itemsets fréquents, l'algorithme génère les règles d'association, en suivant et en restreignant les résultats en fonction de certaines valeurs comme la

Itemsets fréquents	Support
{Pain}	80%
{Lait}	80%
{Couche}	80%
{Bière}	60%
{Pain, Lait}	60%
{Pain, Couche}	60%
{Bière, Couche}	60%
{Lait, Couche}	60%

Tableau 2.2: Itemsets fréquents avec un support minimum de 50%.

confiance. Des exemples de règles d'association sont présentés par le tableau 2.3. Si nous prenons l'itemset fréquent {Bière, Couche} qui dispose de 60% de support (trois des cinq transactions), nous remarquons qu'il est aussi associé à l'article "Lait" dans deux des trois transactions. Nous pouvons donc écrire que {Bière, Couche} $\rightarrow$ {Lait} avec un support de 40% car cette règle apparaît dans deux des cinq transactions et avec une confiance à 67% car {Bière, Couche} apparaîssent trois fois mais ne sont accompagnés par {Lait} que deux fois.

Règles	Support	Confiance
{Lait,Couche} $\rightarrow$ {Bière}	40%	67%
{Lait,Bière} $\rightarrow$ {Couche}	40%	100%
{Bière, Couche} $\rightarrow$ {Lait}	40%	67%
{Bière} $\rightarrow$ {Lait,Couche}	40%	67%
{Couche} $\rightarrow$ {Lait,Bière}	40%	50%
{Lait} $\rightarrow$ {Couche,Bière}	40%	50%

Tableau 2.3: Exemples de règles fouillées depuis le tableau 2.2.

2.3 Règles d'association séquentielles

Bien que les règles d'association classiques aient pu nous apporter des informations pertinentes, nous sommes intéressés par la préservation des séquences d'invocation

de services. De ce fait, nous avons adopté une variante, "*les règles d'association séquentielles*" dans laquelle, A et B deviennent des séquences d'évènements (acheté par un même client, alarmes réseaux, ou tout autre sorte d'évènements généraux). De plus, dans notre cas les séquences suivent un ordre temporel dans le sens où la partie gauche (antécédent) se produit avant la partie droite (conséquent). Les règles qui peuvent être découvertes depuis les traces d'exécution mettent au premier plan des informations cruciales à propos de la chance de voir apparaître des services ensemble dans les traces d'exécution et dans un ordre spécifique. Ainsi, une règle d'association séquentielle peut ressembler à :

$$ServiceA,\ ServiceB\ implique\ ServiceC$$

qui signifie qu'après l'exécution du service A suivi de celle du service B, il y a de bonnes chances de voir le service C apparaître. Dans un souci de clarté, nous avons limité la taille de l'antécédent et du conséquent, néanmoins, les deux côtés de la règle peuvent être des séquences de taille arbitraire.

2.3.1 L'algorithme RuleGrowth

RuleGrowth est un algorithme récent — publié en 2011 — qui vise la découverte de règles d'association séquentielles dans de larges bases de données (Fournier-Viger *et al.*, 2011). Plus spécifiquement, RuleGrowth se focalise sur des séquences composées d'évènements ordonnés dans le temps. Cependant, les évènements peuvent aussi être regroupés en ensemble d'évènements. Dans ce cas ci, les évènements sont considérés comme simultanés.

Dans le tableau 2.4, nous pouvons observer trois séquences différentes composées d'ensembles d'évènements identifiés par des accolades et séparés par des virgules. Lorsque l'on souhaite prédire ce qui va se passer à la suite d'évènements, nous

ID	Items
1	{Clean Code},{Refactoring};{Clean Coder};{Design Patterns, Refactoring to Patterns}
2	{Clean Code, Clean Coder};{Refactoring, Design Patterns};{Head First DP} {Head First DP, Refactoring to Patterns};{Clean Code, Refactoring} {Clean Coder, Refactoring to Patterns};{Design Patterns}
3	{Head First DP};{Patterns of Enterprise Apps};{Clean Code, Refactoring to Patterns} {Refactoring};{Design Patterns}

Tableau 2.4: Base de données de séquences d'achat de livres.

pouvons utiliser les règles d'association séquentielles qui, comme vu dans la section 2.2, sont de la forme $A \rightarrow B$, signifiant que B a des chances de se produire après A. De telles règles ont été utilisées dans de nombreux domaines : l'analyse de cours boursier (Yang *et al.*, 2006) ou l'établissement de prévisions météorologiques (Hamilton, H. J. and Karimi, 2005).

En utilisant les séquences du tableau 2.4, RuleGrowth est capable d'extraire les règles présentées par le tableau 2.5.

Règles
Clean Code ⇒ Refactoring
Clean Code ⇒ Refactoring, Design Patterns
Clean Code ⇒ Design Patterns
Clean Code, Refactoring ⇒ Design Patterns
Clean Code, Refactoring,Clean Coder ⇒ Design Patterns
Clean Code, Refactoring,Head First DP ⇒ Design Patterns
Clean Code, Refactoring, Head First DP, Refactoring to Patterns
Clean Code, Refactoring, Refactoring to Patterns ⇒ Design Patterns
Clean Code, Clean Coder ⇒ Design Patterns
Clean Code, Head First DP ⇒ Design Patterns
Clean Code, Head First DP, Refactoring to Patterns ⇒ Design Patterns

Tableau 2.5: Exemples de règles d'association séquentielles.

RuleGrowth surclasse les autres algorithmes existants – CMRules et CMDeo (Fournier-Viger *et al.*, 2012) – car il utilise des techniques élaborées afin de générer

beaucoup moins de candidats et, par conséquent, être plus performant.

2.4 Conclusion

De nombreuses études ont visé l'extraction de données interprétées depuis les traces d'exécution et certaines d'entre elles ont, avec succès, extrait toutes sortes de patrons pour représenter le système sous analyse. Nous avons proposé une introduction à un sous-domaine de la fouille de données — dont l'étude des traces d'exécution dépend — nommé la fouille de règles d'association et plus particulièrement la fouille de règles d'association séquentielles via deux algorithmes. La fouille de règles d'association classique ou non séquentielle dans les traces d'exécution générées par des systèmes à base de services a déjà été explorée par (Upadhyaya *et al.*, 2012). Dans cette étude, les auteurs proposent une approche pour la découverte de patrons de composition. Néanmoins, ils ne tiennent pas compte de l'ordonnancement des appels ce qui autorise la génération de plus de règles et en réduit donc leur pertinence. La technique de la fouille de règles d'association séquentielles sera utilisée dans notre approche décrite dans le chapitre suivant.

CHAPITRE III

L'APPROCHE SOMAD

Dans ce chapitre, nous présentons l'approche SOMAD (*Service Oriented Mining for Antipatterns Detection*), composée de cinq étapes, pour la détection d'anti-patrons basée sur les traces d'exécution produites par les systèmes à base de services. Cette nouvelle approche est une variante de SODA (Nayrolles *et al.*, 2012; Moha *et al.*, 2012) basée sur les traces d'exécution. Les traces peuvent provenir de n'importe quelles implémentations SOA. SODA est applicable uniquement sur des systèmes SCA en utilisant un ensemble de scénarios prédéfinis. Plus particulièrement, dans SOMAD, nous spécifions un nouvel ensemble de métriques qui s'appliquent sur des règles d'association séquentielles extraites depuis les traces d'exécution. La figure 3.1 montre une vue d'ensemble de SOMAD. Les nouvelles étapes correspondantes à SOMAD sont en gris tandis que celles de SODA sont en blanc. La première étape *(Étape 1.Inférence de métriques)* est supportée par la création d'hypothèses provenant de la description textuelle des anti-patrons SOA. Ces hypothèses sont utilisées dans cette étape pour inférer de nouvelles métriques dédiées à l'interprétation de règles d'association séquentielles. La seconde étape *(Étape 4.Fouille de règles d'association)* a pour but d'extraire les règles d'association des traces d'exécution du système à base de services visé. Les règles d'association séquentielles suggèrent des relations intéressantes entre les services dans de larges amas de traces d'exécution.

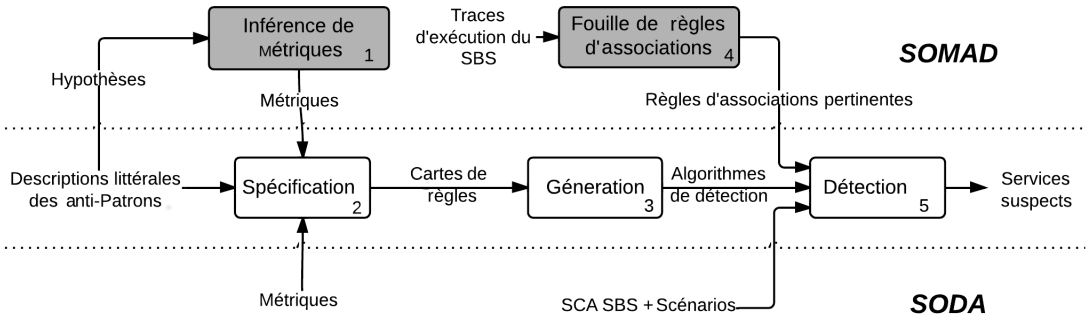


Figure 3.1: Les approches SODA et SOMAD. Les cases grises correspondent aux nouvelles étapes de SOMAD ajoutées aux étapes de SODA en blanc.

3.1 Méthodologie de SOMAD

Dans cette section nous présentons toutes les étapes SOMAD.

3.1.1 Étape 1. Inférence de métriques

Un ensemble de métriques dédiées à l'interprétation de règles séquentielles d'association est inféré depuis trois hypothèses construites via la description textuelle des anti-patrons (tableau 1.1). Tel que présenté par la figure 3.2, cette étape a comme entrée des hypothèses basées sur les descriptions textuelles des anti-patrons et produit des métriques.

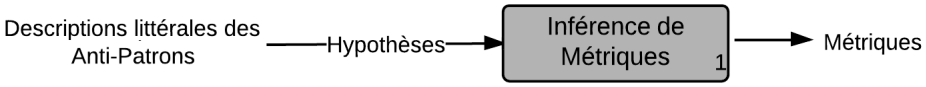


Figure 3.2: Étape 1 : Inférence de métriques.

Ces hypothèses correspondent à des heuristiques qui permettent l'interprétation de

règles d'association séquentielles dans le but d'identifier les propriétés pertinentes des anti-patrons SOA. Après une étude minutieuse des descriptions textuelles des anti-patrons, nous constatons que les anti-patrons SOA peuvent être spécifiés en termes de couplage et de cohésion. Le couplage fait référence au degré de dépendance entre les services tandis que la cohésion représente la cohérence des responsabilités présentées par un service (Perepletchikov *et al.*, 2007, 2010).

Hypothèse 3.1 *Si un service A implique un service B avec un fort support et une grande confiance, alors A et B sont fortement couplés.*

Hypothèse 3.2 *Si un service apparait en tant que conséquent (antécédent) d'un grand nombre de règles d'association séquentielles, alors il a un fort couplage entrant (sortant).*

Les hypothèses 3.1 et 3.2 qualifient le couplage entre deux services et le couplage général en termes de couplage entrant et de couplage sortant.

La cohésion est aussi largement utilisée dans les descriptions textuelles des anti-patrons, de ce fait, nous l'avons prise en compte dans l'hypothèse 3.3.

Hypothèse 3.3 *Si le nombre de méthodes d'un service donné est similaire au nombre de ses différents partenaires (Hypothèse 3.2, le nombre de services avec qui il communique), alors le service n'est pas cohésif.*

En se basant sur les trois hypothèses ci-dessus, nous avons créé des métriques spécifiques au domaine afin d'explorer les manifestations d'anti-patrons SOA cachées dans les règles d'association séquentielles. Afin de mieux comprendre comment nous passons des hypothèses aux métriques, la figure 3.3 donne un exemple pour

le *Tiny Service*. Nous pouvons voir que le Tiny Service est caractérisé par deux de nos trois hypothèses (Hypothèses 3.1 et 3.3) et ces hypothèses ont étés traduites en deux métriques distinctes nommées OC (*Outgoing Coupling*) et NM (*Number of Methods*).

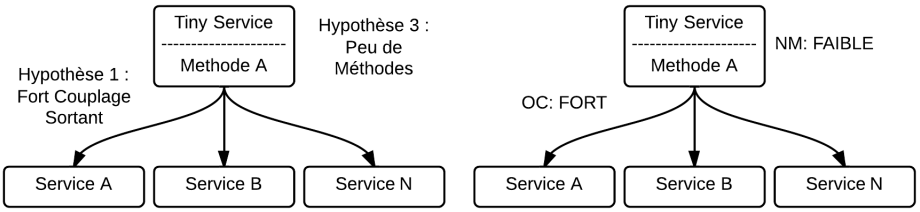


Figure 3.3: Exemple de l'utilisation des hypothèses et métriques : Le *Tiny Service*.

Les métriques que nous avons conçues et implémentées spécifiquement pour supporter SOMAD et qui seront utilisées pour créer des cartes de règles grâce à un DSL (*Domain Specific Language*) sont présentées par les tableaux 3.1 et 3.2. Dans ces tableaux, des notations mathématiques standards ont été utilisées lorsque possible et étendues (ajout de symboles non standards) quand nécessaire. Les règles d'association peuvent être visualisées par (X $\rightarrow$ Y) avec X et Y respectivement l'antécédent et le conséquent d'une règle. K, L sont les services partenaires. AR représente l'ensemble complet des règles d'association, tandis que AR_s et AR_m représente respectivement les sous-ensembles relatifs aux règles d'association au niveau service et au niveau méthode. M_S dénote les méthodes d'un service donné S. Finalement, nous utilisons des symboles non-standards pour les séquences d'opérations : [] est le constructeur de séquences, $\Cup$ signifie concaténé (*append* en anglais) dans la séquence ; $\Subset$ représente les sous-séquences d'une relation ; et A $\lessdot$ B veut dire que le service A apparaît dans la séquence B.

Number of Matches (NMA(S)) : $\#\{X \rightarrow Y \in AR_s \mid S \lessdot (X \Cup Y)\}$ }
Compte le nombre de règles dans lesquelles un service apparaît ; que ce soit en tant qu'antécédent ou conséquent.
Number of Diff. Partners (NDP(S)) : $\#\{K \mid X \rightarrow Y \in AR_s, S \lessdot X, K \lessdot Y\} + \#\{K \mid X \rightarrow Y \in AR_s, S \lessdot Y, K \lessdot X\}$
Indique combien de partenaires a un service. En d'autres mots, cette métrique détermine si le service communique avec beaucoup d'autres services ou non.
Incoming Coupling (IC(S)) : $\sum_{L \in \{K \mid X \rightarrow Y \in AR_s, K \lessdot X, S \lessdot Y\}} \frac{CID(S,X)}{NDP(S)}$
Compte le nombre de fois où un service est utilisé. Cependant, au lieu de compter de manière classique l'apparition du service, nous utilisons une valeur contextuelle : $\frac{CID(S,X)}{NDP(S)}$ où X est le service partenaire. Ainsi, le couplage est inversement proportionnel au nombre de partenaires différents.
Outgoing Coupling (OC(S)) : $\sum_{L \in \{K \mid X \rightarrow Y \in AR_s, S \lessdot X, K \lessdot Y\}} \frac{CID(X,S)}{NDP(S)}$
Les mêmes principes s'appliquent que ceux vus pour l'*Incoming Coupling*. Cette métrique compte le nombre de fois où le service visé utilise d'autres services.
Number of Methods (NM(S)) : $\#\{K \mid X \rightarrow Y \in AR_m, K \in M_s, K \lessdot (X \Cup Y)\}$
Compte le nombre de méthodes différentes exposées dans les règles d'association séquentielles. Cette métrique porte uniquement sur le sous-ensemble AR_m.
Cohesion (COH(S)) : $\frac{NDP(S)}{NM(S)}$
Fournit un ratio comprenant le nombre de partenaires différents et le nombre de méthodes disponibles.

Tableau 3.1: Métriques simples.

Cross Invocation Dependencies (CID(S_a, S_b)***)*** :
$\#\{X \to Y \in AR_s \mid S_a \lessdot X, S_b \lessdot Y\} + \#\{X \to Y \in AR_s \mid S_a \lessdot Y, S_b \lessdot X\}$
Cette métrique est cruciale pour la détection des manifestations enfouies dans les traces. En effet, elle explore les interactions typiques entre les services tout en ignorant les interactions non fréquentes (dû au seuil à atteindre, voir section 2.2). Pour obtenir cette information, CID compte toutes les règles d'association où un service a (S_a) est présent dans la partie des antécédents et un service b (S_b) dans la partie des conséquents ou vice et versa.
Transitive Coupling (TC(S_a, S_b)***)*** :
$\#\{K \mid X \to Y \in AR_s, S_a \lessdot X, S_b \lessdot Y, ([S_a, K] \Subset X \vee [K, S_b] \Subset Y)\}$
Transitive Coupling a été bâtie pour détecter un anti-patron SOA particulier, le *Service Chain* (voir 1.1). Tout d'abord, nous avons observé qu'une paire de services qui ne communiquent pas directement ne veut pas pour autant dire que les services ne sont pas couplés. Ceci est l'idée fondatrice du *Service Chain* et de cette métrique. Afin d'identifier le couplage transitif dans les règles d'association, ce qui est beaucoup plus difficile que le couplage direct, nous avons besoin de représenter une chaîne dans les règles d'association. Un service a (S_a) est dans l'antécédent d'une règle et un service b (S_b) est dans le conséquent d'une autre règle. Ces deux règles sont connectées grâce à un service k (S_k) qui apparait dans le conséquent de la première règle et dans l'antécédent de la seconde règle. Ainsi, dans un cas trivial, nous pourrions avoir :[a]→[b] et [b]→[c]. Dans cette configuration, a et c ne sont pas directement couplés mais si c est indisponible, il y a de bonnes chances pour que a et b le soient également. Des chaines de distance supérieur à 3 sont également possibles.

Tableau 3.2: Métriques complexes.

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

2  rule           ::= RULE:rule_name {content_rule};
3  content_rule   ::= metric | relationship | 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 ordi_value
8                   |metric_value comparator num_value
9  metric_value   ::= id_metric (num_operator id_metric)?
10 num_operator   ::= + | - | * | /

11 id_metric      ::=  NMA | NDP | NM | COH | CID | IC | OC | TC

12 ordi_value     ::= VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW
13 comparator     ::= EQUAL | LESS | LESS_EQUAL | GREATER | GREATER_EQUAL

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

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

Figure 3.4: Grammaire BNF utilisée pour construire les cartes de règles.

3.1.2 Étape 2. Spécification d'anti-patrons SOA

Les métriques définies à l'étape 1 nous permettent de spécifier des anti-patrons SOA sous la forme de carte de règles. Afin de combiner ces métriques, nous avons utilisé un DSL défini par (Moha *et al.*, 2012) et qui a ensuite été perfectionné par (Demange *et al.*, 2013) ainsi que (Nayrolles *et al.*, 2012). Ce DSL est présenté dans la figure 3.4 sous la forme d'une grammaire Backus-Naur. Une règle décrit une métrique, une relation, ou une combinaison d'autres règles (ligne 3) en utilisant un ensemble d'opérateurs (ligne 6). Les métriques qui sont disponibles apparaissent ligne 11. Une métrique peut être définie comme une combinaison d'autres métriques (lignes 9 et 10). Chaque métrique peut être comparée à une ou plusieurs valeurs ordinales (ligne 7) – un ensemble de cinq valeurs de l'échelle de Likert (Michael S. et Jacob, 1971) de très faible (*very low*) à très fort (*very high*) – ou être comparées à une valeur numérique (ligne 8) en utilisant des comparateurs (ligne 13).

La figure 3.5 présente les entrées de cette seconde étape, à savoir les descriptions textuelles des anti-patrons ainsi que les métriques issues de l'étape 1. Cette étape de spécification produit des cartes de règles.

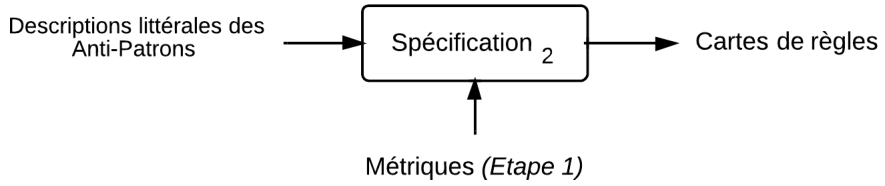


Figure 3.5: Étape 2 : Spécification d'anti-patrons SOA.

Les cartes de règles utilisées pour spécifier les anti-patrons SOA sont présentées par la figure 3.6. A titre d'exemple, la carte de règles correspondant au *Tiny Service* (figure 3.6-b) est composée de trois règles. La première (ligne 2) est une intersection de deux règles (lignes 3 et 4) qui définissent deux métriques : un fort couplage sortant (*Outgoing Coupling OC*) et un faible nombre de méthodes (*Number of Methods NM*).

3.1.3 Étape 3. Génération des algorithmes de détection

Depuis les spécifications des anti-patrons SOA décrites avec la DSL, nous générons automatiquement les algorithmes de détection. Nous utilisons maintenant Ecore (Sciamma *et al.*, 2013) et Acceleo (Obeo, 2005) pour automatiser la génération des algorithmes. La figure 3.7 expose les entrées de cette étape : les cartes de règles générées à l'étape 2 ainsi qu'un gabarit `Java`. Cette étape produit des algorithmes de détection directement exécutables.

Pour la génération automatique des algorithmes de détection, nous commençons par analyser syntaxiquement (*parser* en anglais) les *cartes de règles* pour chaque

```
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

Figure 3.6: Cartes de règles pour nos anti-patrons.

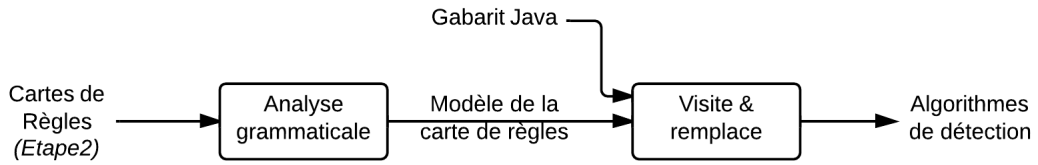


Figure 3.7: Étape 3 : Génération d'algorithmes.

anti-patron et nous les représentons comme des modèles. Par la suite, nous utilisons Ecore pour les valider syntaxiquement par rapport au méta-modèle de notre DSL. Nous utilisons une génération de code automatique basée sur les modèles et fournie par Acceleo (Obeo, 2005). Pour ce faire, nous définissions un modèle unique pour toutes les cartes de règles qui contient des *étiquettes*, qui seront par la suite, remplacées par les métriques définies dans la carte de règles. Finalement, le modèle unique est utilisé pour générer l'algorithme d'une carte de règles et produit une ou plusieurs classes Java, directement compilable et exécutable.

La figure 3.8 montre le modèle du *Multi Service* que nous utilisons pour générer son algorithme de détection. À la première ligne de la figure 3.8, le modèle importe le méta-modèle de notre DSL. Il contient aussi les *tags*, qui sont identifiables par des crochets, et qui correspondent aux variables qui vont être remplacées (*rule card name*, *rule names*, *metrics*, *values* et les différents opérateurs). Un seul modèle est requis pour toutes les cartes de règles. Ainsi, il est aisé de les maintenir.

La figure 3.9 montre le code généré par Acceleo. Ce code est basé sur la carte de règles de la figure 3.8. Cette génération crée une classe Java avec les différents opérateurs et les différentes métriques. La classe générée est directement compilable et exécutable en utilisant un *Java Class Loader*. Les ingénieurs n'ont plus qu'à fournir l'implémentation concrète des métriques qu'ils souhaitent utiliser.

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

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

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

Figure 3.8: Gabarit pour la génération automatique.

```
package com.soda.antipatterns;

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

public class MultiService extends AMotif {

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

}
```

Figure 3.9: Capture d'écran du code généré pour le *Multi-service.* .

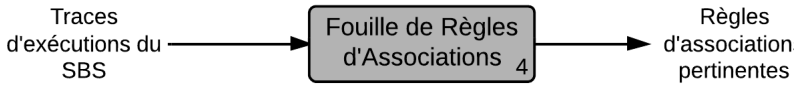


Figure 3.10: Étape 4 : Fouille de règles d'association.

Ce processus est complètement automatisé pour éviter les tâches manuelles qui sont source d'erreurs. Ce processus garanti aussi la traçabilité entre la spécification des anti-patrons SOA faite avec la DSL et la détection qui sera effectuée sur le système à base de services. De ce fait, les ingénieurs logiciels peuvent se focaliser sur la spécification des anti-patrons, sans avoir à considérer les détails techniques de l'aggrégation de règles et métriques.

Nous précisons aussi que, dans cette étape, la génération automatique des algorithmes était déjà présente dans SODA et a été améliorée par Christopher Robert. Nous l'avons juste utilisée afin de créer de nouveaux algorithmes composés de nos métriques et capable de détecter des anti-patrons en interprétant les règles d'association séquentielles.

3.1.4 Étape 4. Fouille des règles d'association

Les traces d'exécution sont analysées pour extraire les règles d'association séquentielles. Ce processus est illustré par la figure 3.10.

Les règles d'association séquentielles sont fouillées depuis une collection de traces d'exécution en utilisant un support et une confiance minimum. Une transaction est un ensemble d'appels de services et de méthodes ordonnés dans le temps.

Nous rappelons que le support d'un patron [1], c'est-à-dire, d'une séquence d'items (appels de services ou méthodes), est le pourcentage global de transactions qui contiennent ce patron. Quant à la confiance, elle mesure la vraisemblance que la partie conséquente soit après la partie antécédente dans une même transaction.

Pour extraire les règles d'association séquentielles, deux possibilités s'offraient à nous. D'une part, la fouille de patron séquentiel où les algorithmes de fouille de règles ont été créés pour des structures légèrement plus générales que celles utilisées ici. Dans les faits, les patrons séquentiels sont définis par des transactions qui représentent des séquences d'ensemble. Des travaux intéressants pour la découverte de patrons séquentiels ont été publiés, e.g, la méthode PrefixSpan (Mortazavi-Asl *et al.*, 2004). D'autre part, les traces d'exécution ne contiennent pas réellement de pures transactions séquentielles. En effet, leur structure sous-jacente est composée d'éléments individuels. De telles données sont connues depuis le milieu des années 90, cependant, elles ont reçu moins d'attention de la part de la communauté de la fouille de données ; sans doute parcequ'elles sont moins intéréssantes à fouiller. Néanmoins, de nombreuses applications pratiques ont été imaginées dans des domaines où ce genre de données prolifère, notamment dans la fouille de traces d'exécution de logiciel. Dans la littérature générale de la fouille de données, fouiller des séquences pures, en opposition à la fouille de séquence composées d'ensembles, a été traitée par la fouille d'épisodes (Mannila *et al.*, 1997). Les épisodes sont composés d'évènements dans le sens où un appel de service est un évènement. Sans aucun doute, le domaine disposant de la plus grande base de connaissance sur le sujet est la fouille d'utilisation de site web. Les données d'entrées sont à nouveau des traces d'exécution, néanmoins cette fois les traces sont des requêtes envoyées à un serveur web (Pei *et al.*, 2000). Il est

1. Dans le contexte des règles d'association ; le mot *patron* représente un motif dans les traces d'exécution et non un patron de conception logiciel.

important de souligner que les patrons séquentiels sont plus généraux que ceux basés sur des séquences pures. En effet, les algorithmes de fouille qui ont été créés pour les premiers peuvent être moins efficace que ceux créés pour les seconds car des étapes supplémentaires peuvent être requises pour lister les ensembles significatifs. Néanmoins, nous avons choisi d'utiliser un algorithme de fouille de patrons et règles séquentielles qui, malgré les spécificités de nos données — composées d'épisodes et non de séquences — fût performant. Nous avons utilisé tout d'abord l'algorithme RuleGrowth (Fournier-Viger *et al.*, 2011) qui semblait être le plus adapté et a l'avantage d'être disponible gratuitement [2]. Bien que non optimisé pour les séquences pures, ses performances sont plus que satisfaisantes. Cependant, dans le prochain chapitre, nous présenterons une évolution de RuleGrowth nommée SOARuleGrowth qui est plus adaptée à nos données. Pour résumer, à la fin de cette étape, nous avons extrait les relations pertinentes entre les services. Ces relations ont la forme de règles d'association séquentielles.

La figure 3.11 présente le processus de fouille de règles d'association séquentielles dans les traces d'exécution appliqué au *Tiny Service*. Dans les traces d'exécution, on constate que le Service A ne dispose que d'une seule méthode, la méthode A. De plus cette méthode fait appel aux services B, C ou D. Ainsi, les règles d'association générées pour cet ensemble de traces factices sont : A $\rightarrow$ B, A $\rightarrow$ C et A $\rightarrow$ D.

3.1.5 Étape 5. Détection d'anti-patron SOA

Cette dernière étape consiste à appliquer les algorithmes de détection générés à l'étape 3 sur les règles d'association séquentielles fouillées à l'étape 4. A la fin de cette étape, les services du système suspectés d'être impliqués dans un anti-patron

2. `http://www.philippe-fournier-viger.com/spmf/`

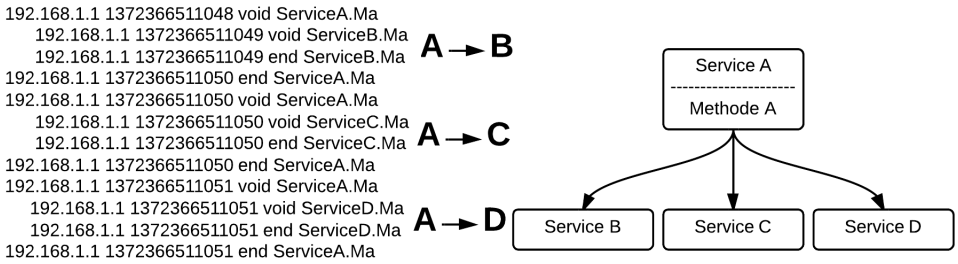


Figure 3.11: Fouille de règles d'association depuis les traces d'exécution : *Le Tiny Service*.

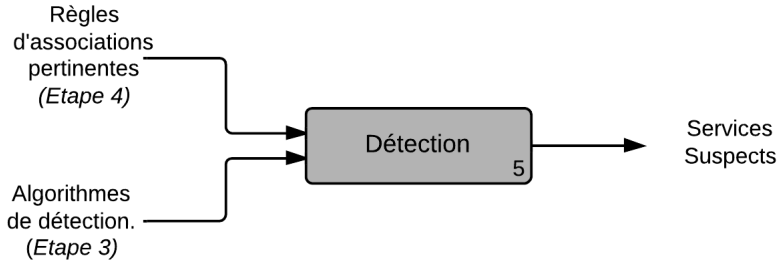


Figure 3.12: Étape 5 : Détection d'anti-patrons SOA.

SOA sont identifiés, comme présenté à la figure 3.12.

Le processus de détection est illustré dans la figure 3.13. On s'aperçoit que toutes les règles d'association générées depuis les traces d'exécution de la figure 3.11 impliquent une seule et même méthode : `SA.mA`. De plus, cette méthode communique avec trois services distincts : B, C et D. De ce fait, les resultats des métriques NM (*Number of Methods*) et OC (*Outgoing Coupling*) seront respectivement 1 et 3. Si ces valeurs sont jugées statistiquement faibles et hautes sur l'échelle de Likert respectivement, alors le service A sera identifié comme un *Tiny Service*, comme indiqué dans la carte de règles présentée par la figure 3.6-b.

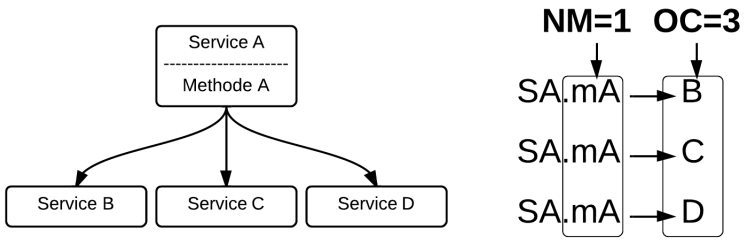


Figure 3.13: Détection d'un *Tiny Service*.

Au cours de ce chapitre, nous avons présenté en détail les cinq étapes composant l'approche SOMAD pour la détection automatique d'antipatrons dans les applications à base de services. Nous avons aussi présenté les hypothèses, métriques et processus de génération d'algorithme utilisés pour cette détection. Dans le prochain chapitre, nous présentons l'implémentation qui supporte l'approche SOMAD.

CHAPITRE IV

IMPLÉMENTATION DE SOMAD

Dans ce chapitre, nous présentons l'implémentation qui supporte l'approche SOMAD. Nous présentons aussi les raisons techniques qui font en sorte que SOMAD a de meilleurs résultats que SODA, en termes de précision et performances. Nous allons tout d'abord étudier la génération des traces d'exécution puis leur collecte et agrégation. Enfin, nous nous attarderons sur l'identification des transactions dans ces traces et la modification de l'algorithme RuleGrowth pour l'adapter à nos besoins.

4.1 Génération de traces d'exécution

Dans le cas où les traces ne sont pas disponibles, cette étape permet leur génération. Si le système à base de services cible ne produit pas de traces d'exécution qui contiennent toutes les informations requises, nous devons l'instrumentaliser dans ce but. De telles traces permettent de déboguer les applications quand des débogueurs ne sont pas disponibles ou applicables (souvent le cas dans les environnements SOA). La production de traces d'exécution peut, cependant, introduire de l'obfuscation [1] de code. Néanmoins, elle peut aussi avoir des bénéfices au niveau

1. Le code impénétrable ou offusqué d'un programme informatique est un code dont la compréhension est difficile pour un humain tout en restant parfaitement compilable par un ordina-

de la compréhension de l'architecture car le code doit être parfaitement maîtrisé afin d'être instrumentalisé correctement. Cette technique de production de traces d'exécution est la plus commune. Cependant, si le code source n'est pas disponible, une autre technique consiste à instrumentaliser l'environnement d'exécution du système à base de services cible. Par exemple, LTTng (Fournier et Dagenais, 2009) instrumentalise les systèmes de type Linux afin qu'ils produisent des traces d'exécution avec un faible sur-coût en termes de temps d'exécution.

Pour faciliter le traitement automatique des traces d'exécution, nous avons imaginé un modèle (voir figure 4.1) qui est un bon compromis entre simplicité et quantité d'informations. Dans ce modèle, une invocation de méthode génère deux lignes, une ligne d'ouverture et une ligne de fermeture avec l'identification d'un client correspondant à son @IP et un marqueur temporel (*timestamp*). La présence de deux lignes est nécessaire pour identifier des appels à d'autres services avant la fin de la méthode. En effet, les traces contiendront une nouvelle ligne d'entrée dans une méthode avant la ligne de fermeture de la méthode initiatrice.

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

Figure 4.1: Modèle de trace.

Les systèmes à base de services contiennent souvent des sous-systèmes de génération de traces d'exécution. Cependant, ces systèmes déjà intégrés peuvent produire des traces très différentes de notre modèle. En conséquence, nous avons

---

teur.

rendu SOMAD adaptable par un simple DSL (*Domain Specific Language*) basé sur les expressions régulières (voir figure 4.2). De ce fait, les informations peuvent être ordonnées différemment.

```
time{^\w+\s\d\d\s\ :\d\d :\d\d.\d+} end{end}
method{^[^.]*.(.*)$} service{$[^.]*.(*.)}
customer{\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1, 3}\b}
line{ *customer *time *(end)? *method.service *}
```

Figure 4.2: DSL associé au modèle de la figure 4.1 .

4.2 Collecte des traces d'exécution et agrégation

Le but ici est de télécharger les fichiers de traces là où elles se trouvent puis de les agréger dans un seul et même fichier.

Les traces d'exécution sont générées par les services composant le système, puis elles sont collectées et agrégées. Cette étape est importante – elle construit les données d'entrée de SOMAD – et non triviale (Wilde *et al.*, 2008). En effet, la nature hautement dynamique et distribuée des systèmes à base de services introduit deux défis distincts. Le premier est lié à la distribution des systèmes à base de services et, donc, des traces d'exécution. En effet, chaque service génère ses traces d'exécution dans son environnement propre. De ce fait, nous devons connaître l'endroit où s'exécutent les services et chaque environnement doit disposer d'un mécanisme pour télécharger les traces d'exécution. Le second défi est lié au dynamisme ; en effet, les services peuvent être consommés par de nombreux clients en même temps, et de ce fait, les traces d'exécution peuvent s'entrelacer. Afin de résoudre ces problèmes, nous utilisons une approche basée sur celle de (Yousefi et Sartipi, 2011) :

— Nous téléchargeons les traces d'exécution distribuées dans l'architecture du système à base de services en utilisant une base de connaissances préalable. Cette base de connaissances contient les dépôts où aller chercher les traces d'exécution, les protocoles, et toute autre information utile.
— Nous rassemblons les différents fichiers dans un fichier unique.
— Nous trions les traces d'exécution en utilisant leurs marqueurs temps. Cette étape nécessite que les différents environnements d'exécution des différents services soient synchronisés sur la même horloge. En règle générale, cette contrainte est facilement atteignable car les organisations peuvent se synchroniser avec des services d'horloge externe.
— Nous exploitons les relations appelant-appelé entre services et méthodes pour distinguer des blocs de transactions concurrentes.
— Nous classons les transactions par client (IP).

4.3 Identification des transactions

Comme exposé dans les chapitres précédents, une transaction est une séquence d'appels de services et de méthodes. Parmi les transactions identifiées dans les traces d'exécution, nous nous focalisons uniquement sur les transactions contenant plus d'un service. En effet, notre but étant d'identifier des conceptions de faible qualité, un appel unique ne peut que difficilement nous renseigner sur l'architecture sous-jacente. La figure 4.3 présente des traces d'exécution indentées [2] par transaction. Une fois les transactions triviales — contenant un appel — retirées de l'ensemble des transactions, deux tables de transactions — et donc de règles d'association séquentielles — sont générées. La première des deux tables

2. L'indentation ne fait pas partie du format. Elle a été introduite ici pour simplifier la lecture.

```
 1 192.168.1.102 1372366511048 public abstract void serviceA.methodA();
 2         192.168.1.102 1372366511050 public abstract void serviceB.methodB();
 3                 192.168.1.102 1372366511052 public abstract void serviceC.methodC();
 4                 192.168.1.102 1372366511052 exit public abstract void serviceC.methodC();
 5         192.168.1.102 1372366511060 exit public abstract void serviceB.methodB();
 6         192.168.1.102 1372366511065 public abstract void serviceD.methodD();
 7         192.168.1.102 1372366511080 exit public abstract void serviceD.methodD();
 8         192.168.1.102 1372366511082 public abstract void serviceD.methodD2();
 9                 192.168.1.102 1372366511090 public abstract void serviceE.methodE();
10                 192.168.1.102 1372366511095 exit public abstract void serviceE.methodE();
11         192.168.1.102 1372366511100 exit public abstract void serviceD.methodD2();
12 192.168.1.102 1372366511110 exit public abstract void serviceA.methodA();
```

Figure 4.3: Traces d'exécution indentées par transaction.

est au niveau des services, tandis que la seconde est au niveau des méthodes. Ainsi, à chaque fois qu'une méthode est invoquée, la première table enregistre les services impliqués dans cet appel alors que la seconde enregistre les méthodes impliquées. Générer deux tables à deux niveaux de granularité différent améliore la performance de nos algorithmes. En effet, la première table à haute granularité — services — est générée en premier, et les algorithmes de détection sont appliqués à ce niveau. Ensuite, l'investigation est poussée au second niveau de granularité — les services + les méthodes — uniquement pour les services désignés comme suspects à la fin de la première passe d'analyse. Durant la première phase, nous générons plus vite nos règles d'association séquentielles car il y a moins d'objets, et durant la seconde, nous traitons moins de traces puisque les services non-suspects sont ignorés. Le cumul des deux traitements s'est révélé plus performant qu'un traitement à granularité fine – méthodes + services – uniquement.

Transaction	Services participants	Méthodes participantes
1 (Lignes 1 à 12)	(A, B, C, D, D, E)	(A, B, C, D, D2, E)
2 (Lignes 2 à 5)	(B, C)	(B, C)
3 (Lignes 8 à 11)	(D, E)	(D2, E)

Tableau 4.1: Extraction désirée.

Le tableau 4.1 présente les transaction extraites par services et par méthodes pour l'exemple de la figure 4.3. Nous y présentons les deux tables de transactions.

Afin d'obtenir le résultat présenté par le tableau 4.1, nous avons conçu un algorithme dédié. Cet algorithme (Algorithme 4.1) a pour seul but de parcourir les traces d'exécution afin d'en extraire des séquences qui seront par la suite utilisables par les algorithmes de fouille de règles d'association séquentielles. Cette algorithme vise l'extraction des différentes transactions impliquant plus d'un service.

L'intuition derrière cet algorithme est plutôt simple. En effet, pour chaque ligne dans les traces d'exécution, il détermine si la ligne est un début ou une fin de transaction. Si la ligne en cours est un début de transaction alors la position de l'algorithme dans les traces est sauvegardée puis nous cherchons la fin de la transaction en cours. Lorsque la fin est trouvée, toutes les opérations entre le début et la fin sont ajoutées à une nouvelle transaction. Comme les transactions peuvent être imbriquées, l'algorithme retourne à la position qu'il avait sauvegardée et cherche un nouveau début de transaction. Ce processus est répété jusqu'à la fin des traces d'exécution.

Cet algorithme se compose de nombreuses boucles imbriquées. La première s'étendant des lignes 2 à 20 permet simplement de dérouler les traces d'exécution jusqu'à la fin. A l'intérieur de cette boucle, chaque trace κ est analysée (ligne 4) afin de déterminer si c'est une trace de sortie (qui contient le mot clefs `exit`). Si nous sommes en présence d'une trace d'entrée alors, cette trace est ajoutée à la transaction courante (ligne 6) et notre position dans l'ensemble des traces est sauvegardée (ligne 5). La prochaine opération consiste à trouver la fin de cette transaction. Ceci est réalisé par la boucle `tant que` s'étendant des lignes 7 à 12. A l'intérieur de cette boucle, nous continuons à progresser dans l'ensemble des transactions et ajoutons les lignes ne contenant pas d'`end` à la transaction courante (ligne 10). Nous progressons tant que la ligne courante ne correspond pas à la sortie de la transaction courante (ligne 7). Lorsque les lignes κ sont ajoutées

à la transaction courante, en réalité, une ligne est ajoutée dans la transaction courante par service et une autre dans la transaction courante par méthode. La seule différence entre ces lignes est leur formatage. En effet, pour les méthodes, la ligne est gardée entière, tandis que pour les services, elle est tronquée pour ne conserver que le nom du service. Finalement, si la transaction n'est pas triviale, c-à-d, qu'elle a une taille supérieure à 1 (supérieur à 1 service), nous l'ajoutons à l'ensemble des transactions (ligne 14). Nous explorons ensuite la suite des transactions en replaçant notre index à une ligne après le début de la transaction que nous venons d'identifier à la ligne 16.

Données: Traces d'exécution

Résultat: Transactions

```
1  pour chaque ligne κ dans les traces faire
2      si κ n'est pas une trace de sortie alors
3          débutDeTransaction ← positionCourante;
4          κ ← transactionCourante;
5          tant que κ_positionCourante ≠ κ_debutDeTransaction faire
6              positionCourante++;
7              si κ_positionCourante n'est pas une trace de sortie alors
8                  κ_positionCourante ← transactionCourante;
9              fin
10         fin
11         si taille de transactionCourante ≠ 1 alors
12             transactionCourante ← Transactions;
13         fin
14         positionCourante ← débutDeTransaction;
15     sinon
16         positionCourante++;
17     fin
18 fin
```

Algorithme 4.1: Extraction de transactions depuis des traces d'exécution formatées.

4.4 Adaptation de RuleGrowth pour la détection d'anti-patrons : SOARuleGrowth

L'algorithme RuleGrowth (Fournier-Viger *et al.*, 2011) est un algorithme qui permet de fouiller des règles d'association séquentielles dans de grand ensemble d'évènements qui peuvent être simultanés. Bien que performant en termes de temps d'exécution et de précision, l'algorithme RuleGrowth n'est pas tout à fait adapté à nos données. Dans les sous-sections suivantes, nous présentons l'algorithme RuleGrowth et les améliorations que nous lui avons apportés afin qu'il soit mieux adapté à nos données et d'augmenter notre précision.

4.4.1 RuleGrowth

L'algorithme 4.2 expose les principes généraux de l'algorithme de RuleGrowth. Cet algorithme commence par différencier les éléments fréquents et les éléments non-fréquents. Ensuite, l'algorithme détermine si deux élèments juxtaposés possèdent les valeurs nécessaires, en termes de support et de confiance, pour la génération d'une règle. Enfin, si une règle est générée, elle peut être étendue en y ajoutant de nouveaux candidats à droite ou à gauche.

Ces opérations nommées *Chercher de nouveaux éléments à gauche* et *Chercher de nouveaux éléments à droite* sont des tentatives d'expansion de la règle d'association à gauche et à droite. Dans les deux opérations d'expansion, l'algorithme considère une règle ne possédant qu'un élément de chaque coté (A→B) puis parcourt les éléments sur la gauche (droite) de l'élément gauche (droit) des transactions dans lesquelles la règle (A→B) apparaît.

Si la règle d'origine agrémentée d'un nouvel élément à gauche (droite), est toujours supérieure aux minimums de confiance et de support requis, alors *Chercher de*

nouveau éléments à gauche (*Chercher de nouveaux éléments à droite*) est rappelée de manière récursive avec une règle de la forme KA→B (A→BK) afin de continuer l'expansion.

Les éléments qui ne sont pas fréquents — qui n'apparaissent pas assez pour atteindre le seuil support fixé — sont retirés de l'ensemble des séquences (ligne 1). Ensuite, les éléments restants sont ajoutés à une liste d'éléments fréquents lors d'une boucle s'étendant des lignes 2 à 5. À la fin de ces deux premières opérations, nous avons une liste d'éléments sur lesquels travailler afin de générer des règles d'association séquentielles. Par la suite, nous trouvons une double boucle dans laquelle l'algorithme détermine combien de partenaire communs ont deux éléments qui sont juxtaposés (lignes 7 à 24). Dans la seconde boucle — celle qui est imbriquée — si le nombre de partenaires identifiés est supérieur au seuil, alors les règles sont générées (IJ, ligne 15 et JI, ligne 20). Finalement, l'algorithme tente d'étendre la règle générée en cherchant des éléments potentiels à droite et à gauche (lignes 16, 17, 21, 22).

```
1  Retirer les élèments qui ne sont pas fréquents;
2  pour tous les elements κ restant dans Sequences faire
3      si Support de κ ≥ Support minimum alors
4          κ ← ElementsFrequents;
5      fin
6  fin
7  pour tous les élèments κI dans ElementsFrequents faire
8      OccurrenceI ← compter nombre de κ_I;
9      PartenairesI ← Trouver partenaire dans OccurrenceI;
10     pour tous les elements κJ dans ElementsFrequents à partir de position courante +
       1 faire
11         OccurrenceJ ← compter nombre de κJ;
12         PartenairesJ ← Trouver partenaire dans OccurrenceJ;
13         Construire une liste de partenaires communs à I & J;
14         si le nombre de partenaires communs pour IJ est supérieur au support
           minimum alors
15             Générer la règle IJ;
16             Chercher de nouveaux éléments à gauche;
17             Chercher de nouveaux éléments à droite;
18         fin
19         si le nombre de partenaires communs pour JI est supérieur au support
           minimum alors
20             Générer la règle JI;
21             Chercher de nouveaux éléments à droite;
22             Chercher de nouveaux éléments à gauche;
23         fin
24     fin
25 fin
```

Algorithme 4.2: Algorithme RuleGrowth simplifié.

4.4.2 Motivations et changements

Malgré les performances affichées de *RuleGrowth* et les excellents résultats que nous avons obtenus en l'utilisant — 100% rappel et une précision supérieure à celle de SODA d'une marge allant de 2.6% à 16.67% — cet algorithme possède des limitations — liées à nos données — qui doivent être comblées afin d'extraire le maximum de connaissances des traces d'exécution. Afin de combler ces limitations, nous avons effectué trois changements majeurs sur l'algorithme originel décrits ci-bas.

Changement 4.1 *Définir une fenêtre temporelle de travail*

RuleGrowth ne possède pas de fenêtre temporelle définissable. En effet, comme montré par l'exemple d'introduction du Chapitre 2, les séquences et leurs items ne sont pas horodatés. De ce fait, si un utilisateur fait une pause significative dans son utilisation du SBS, les services invoqués avant cette pause pourront être associés avec les services invoqués après cette pause et inversement.

Dans le but de ne pas biaiser les associations, il est impératif de délimiter une fenêtre temporelle dans laquelle différents appels peuvent être associés. Les séquences acceptées devront donc avoir la forme suivante :

```
timestamp{a}, timestamp{b}
```

où `timestamp` sera remplacé par le temps Unix ou Posix. Cette notation décrit un instant dans le temps comme le nombre de seconde écoulées depuis le 1 Janvier 1970 et `a` et `b` des items. A titre d'exemple, `1386197722{a}` représente l'invocation du service A le 5 décembre à 22H55 et 22 secondes. `1386197723{b}` est l'invocation du service b une seconde plus tard.

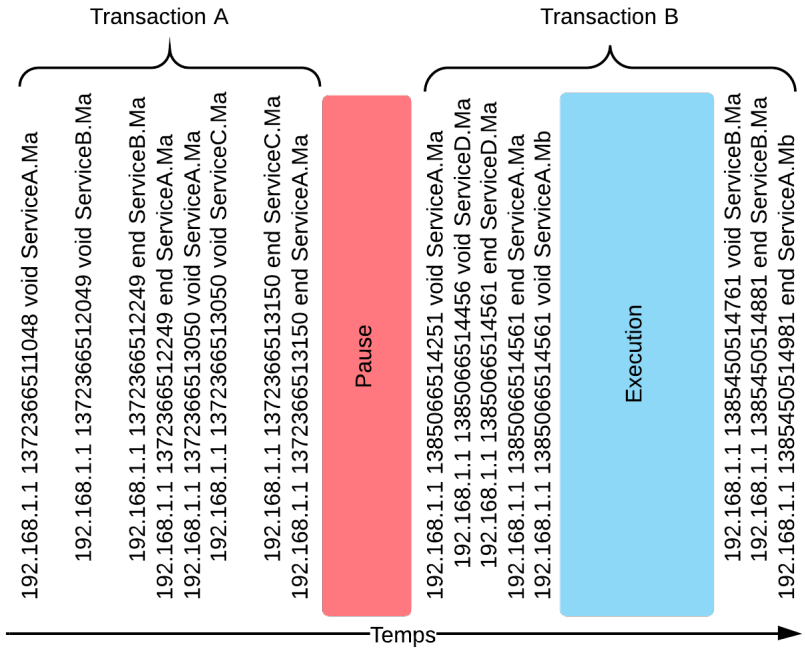


Figure 4.4: Exemple de fenêtres de temps.

Afin d'obtenir le comportement désiré, nous avons modifié la seconde boucle `Pour` de l'algorithme 4.2 par une boucle `Tant que` qui s'achève lorsqu'il n'y a plus d'éléments à parcourir ou lorsque la fenêtre temporelle séparant deux éléments devient supérieure à celle indiquée par l'utilisateur. Néanmoins, comme présenté à la figure 4.4, il faut faire la différence entre des traces qui sont temporellement distantes les unes des autres à cause d'une pause dans l'utilisation et celles qui sont distantes dans le temps à cause du temps d'exécution d'une méthode. Le cas particulier d'un arrêt imprévu du système en cours de transaction (*crash*) est géré par l'algorithme d'extraction de transaction 4.1. De la même manière, les fonctions *Chercher de nouveaux éléments à gauche* et *Chercher de nouveaux éléments à droite* présentes dans l'algorithme 4.2 aux lignes 16, 17 et 21, 22, respectivement, ont été modifiées pour prendre en compte la fenêtre temporelle.

```
1  pour tout κ_I ∈ Elements_Frequents faire
2      Occurrence_I ← compter nombre de κ_I;
3      Partenaires_I ← Trouver partenaire dans Occurrence_I;
4      κ_J ← Elements_Frequents à la position κ_I + 1;
5      tant que κ_J ≠ ∅ & (MarqueurTemps_κJ - MarqueurTemps_κI ≤ Fenêtre_max)
       faire
6          OccurrenceI ← compter nombre de κI;
7          PartenairesI ← Trouver partenaire dans OccurrenceI;
8          pour tout κJ ∈ Elements_Frequents à partir de position_courante+1 faire
9              Occurrence_J ← compter nombre de κJ;
10             Partenaires_J ← Trouver partenaire dans OccurrenceJ;
11             Construire une liste de partenaires communs à I & J;
12             si le nombre de partenaires communs pour IJ est > à supp_min alors
13                 Générer la règle IJ;
14                 Chercher de nouveaux éléments à gauche et à droite;
15             fin
16             si le nombre de partenaires communs pour JI > à supp_min alors
17                 Générer la règle JI;
18                 Chercher de nouveaux éléments à gauche et à droite;
19             fin
20         fin
21         κ_J ← ElementsFrequents a la position κ_J+1;
22     fin
23 fin
```

Algorithme 4.3: Modifications relatives au changement 4.1.

Changement 4.2 *Regrouper des appels quasi-simultanés*

Les transactions extraites par l'algorithme 4.1 sont uniquement composées d'éléments — invocations — simples. Du point de vue de RuleGrowth, cela signifie que les invocations ne sont pas simultanées. Bien que n'étant pas exactement simultanés, il peut être pertinent de regrouper des évènements *presque* simultanés en un unique élément. Grâce à ce regroupement nous pourrons, par exemple, identifier des méthodes dont la première action est d'en appeler une autre et augmenter le poids de la relation (couplage) entre ces méthodes.

```
IP timestamp void methodA.ServiceA();
IP    timestamp void methodB.ServiceB();
IP    timestamp end void methodB.ServiceB();
//    Traiement de la méthode A
//    Invocation des méthodes C, D et E entrecoupées
de traitements.
IP timestamp end void methodA.ServiceA();
```

Figure 4.5: Appels quasi-simultanés.

dispose d'une extraction identique à : `(A,B)(C)(D)(E)` plutôt que `(A)(B)(C)(D)(E)`. De cette manière nous renforçons la relation entre A et B. Une phase de prétraitement doit être ajoutée afin que si `timestamp{a}, timestamp{b}` possèdent un `timestamp` égal ou très proche, alors la séquence soit transformée en `timestamp{a, b}`. Cet ensemble sera désormais considéré comme une suite de deux évènements simultanés. L'algorithme 4.4 présente le prétraitement nécessaire sur les données avant de lancer l'algorithme *SOARuleGrowth*. De la même manière que présenté précédemment par l'algorithme 4.3, nous avons mis en place une boucle `Tant Que` qui prend fin lorsqu'il n'y a plus d'élément ou lorsque la

fenêtre temporelle relative au rassemblement d'évènements est dépassée (lignes 5 à 7). Néanmoins, à l'intérieur de cette boucle, les éléments satisfaisant les conditions sont ajoutés à une liste d'éléments nommée composition (ligne 6), puis les éléments ajoutés sont supprimés de la liste principale d'éléments (ligne 8). Si la composition ainsi créée dispose d'éléments (ligne 10), alors nous rajoutons l'élément duquel nous étions parti à la tête de la composition et le supprimons de la liste principale (ligne 11 et 12). Enfin, nous ajoutons la composition nouvellement créée à la liste principale à la position courante (ligne 13).

```
1  pour tous les Éléments faire
2      κ_J ← Élément à la position κ_I + 1;
3      tant que κ_J ≠ ∅ & (MarqueurTemps_κJ - MarqueurTemps_κI ≤ Fenêtre_max)
       faire
4          κ_J ← composition;
5          Supprimer κJ dans éléments;
6          κ_J ← Élément à la position κJ + 1;
7      fin
8      si Taille de la composition ≻ 0 alors
9          Placer κ_I à la tête de la composition;
10         Supprimer κ_I dans éléments;
11         composition ← éléments à la position κ_I;
12     fin
13 fin
```

Algorithme 4.4: Modifications relatives au changement 4.2.

Changement 4.3 *Éliminer les appels redondants*

Lors de nos tests préliminaires où nous avons utilisé la version originale de *RuleGrowth*, une proportion significative de faux positifs provenaient des appels de services à leurs propres méthodes. Ainsi, si un `ServiceA` invoque sa propre méthode B au cours de l'invocation — par un client externe ou un autre service — de sa méthode A, alors des règles de type : `ServiceA.methodA` $\rightarrow$ `ServiceA.methodB` pouvaient être générées et fausser les métriques de couplage et de dépendance. En effet, l'algorithme ne doit pas considérer comme pertinent les sous-séquences de type : `{ServiceA.methodA, ServiceA.methodB, ServiceA.methodC}` car elles augmentent artificiellement le couplage du service en introduisant A comme partenaire de lui-même. Afin d'introduire ce comportement spécifique dans l'algorithme *RuleGrowth*, nous avons imaginé l'algorithme 4.5. A chaque fois qu'une règle est sauvegardée, nous vérifions si deux éléments juxtaposés dans cette règle appartiennent au même service (ligne 3). Si tel est le cas, le second élément est supprimé de la règle (ligne 4).

4.4.3 Impacts des modifications

Afin de mesurer l'impact de ces modifications, nous avons comparé les deux algorithmes en termes de nombres de règles générées, de longueur moyenne des règles, le temps et de la mémoire nécessaire à leur génération. Nous avons testé les algorithmes en utilisant les données utilisées pour nos expérimentations du prochain chapitre. Le figure 4.6 expose le nombre de règles à des supports fixés (60%, 40%, 20% et 10%). Le nombre de règles générées par *SOARuleGrowth* est inférieur de 36% pour un support égal à 10%. Cette différence significative est due aux changements interdisant de sauvegarder une règle composée d'éléments répartis avant et après une pause ainsi qu'au regroupement d'appels presque simultanés — ré-

```
1  pour tous les élèments d'une règle d'association faire
2      si (κ_J ← κ_I + 1) ≠ ∅ alors
3          si Service_κJ est egal à Service_κI alors
4              Supprimer κ_J de la règle d'association;
5              Calculer le support et la confiance de la règles sans κ_J;
6              si Support ou Confiance inférieurs aux seuils fixés alors
7                  Supprimer la règle;
8              fin
9          fin
10     fin
11 fin
```

Algorithme 4.5: Modifications relatives au changement 4.3.

duisant le nombre d'éléments, et donc, le nombre de possibilités pour créer des règles.

Notre seconde mesure concerne la longueur moyenne des règles générées. Le graphe 4.7 expose les différences entre *SOARuleGrowth* et *RuleGrowth*. Le changement 4.3 spécifie l'élimination des règles qui augmentaient artificiellement le couplage d'un service. De plus, un patron $ServiceA.methodA \Rightarrow Service.A.methodB$ sera aussi éliminé. En conséquence, la longueur moyenne des règles d'association diminue légèrement.

Le graphique 4.8 présente l'impact de nos modifications en termes de mémoire nécessaire. On constate que *SOARuleGrowth* est moins gourmand que son prédécesseur lorsque le support minimum à atteindre est haut car il traite moins d'éléments et donc moins d'éléments fréquents et de règles d'association. Cependant, la supériorité s'inverse pour un support à 10%. Ceci est principalement est lié au fait que (1) les éléments traités par SOARuleGrowth sont plus gros, ils

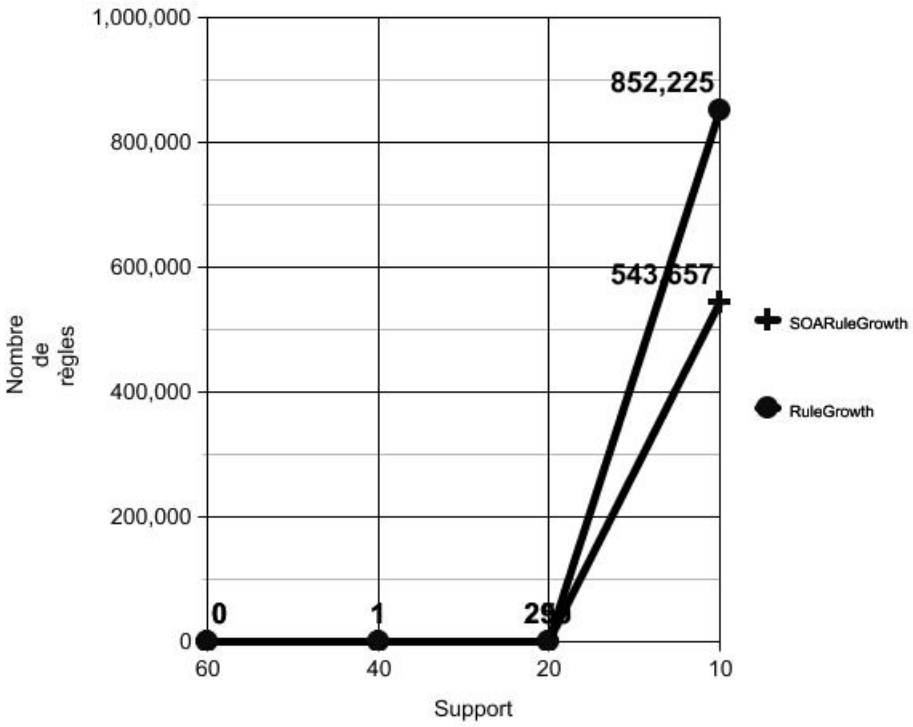


Figure 4.6: Nombre de règles.

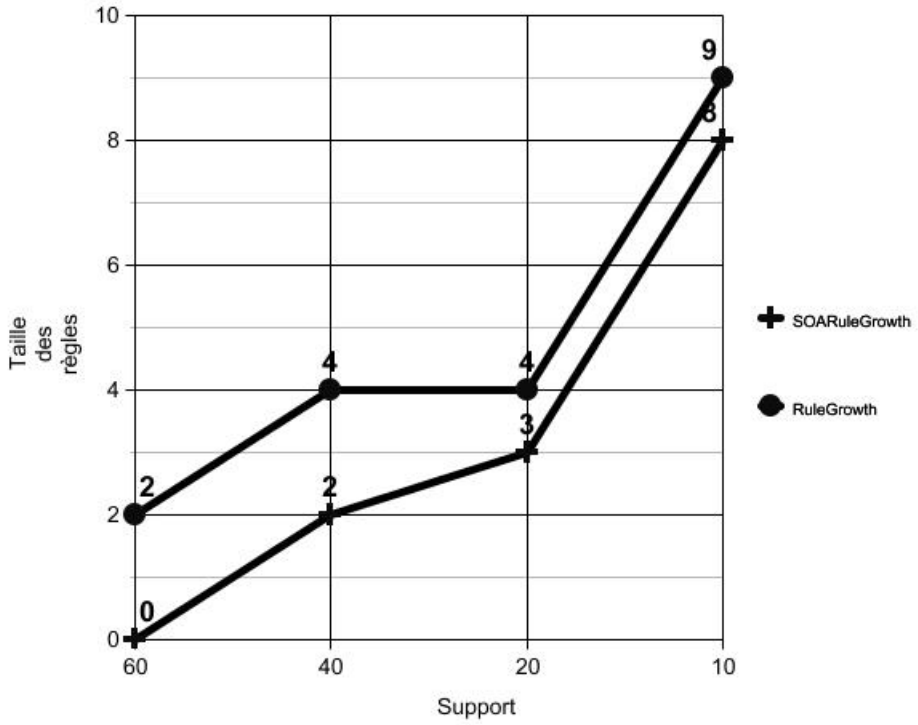


Figure 4.7: Longueur moyenne des règles d'association.

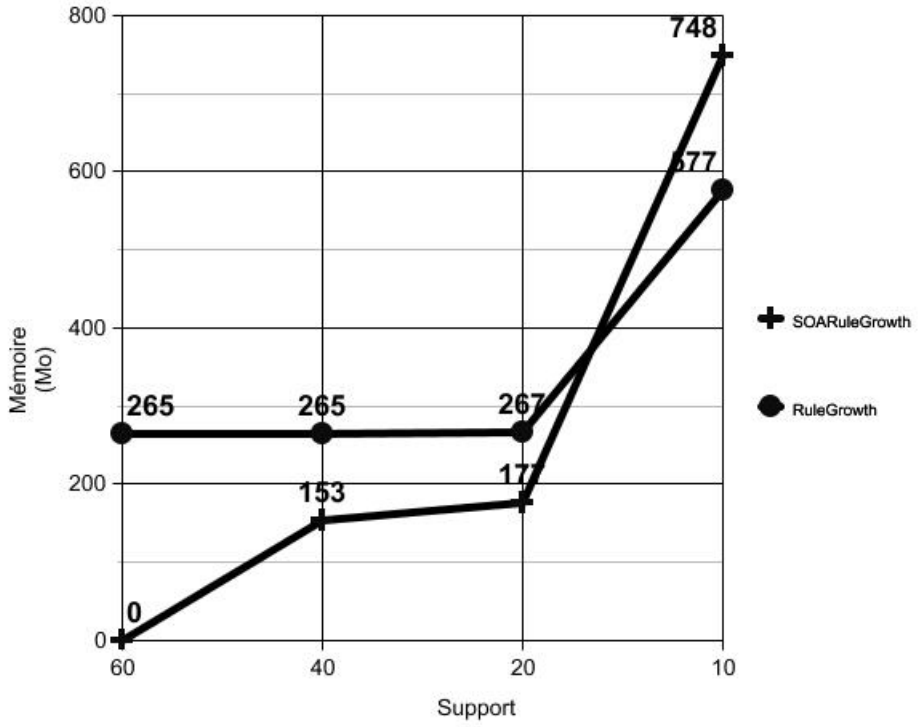


Figure 4.8: Mémoire requise.

contiennent l'horodatage et (2) le temps requis pour le pré-traitement nécéssaire à *SAORuleGrowth* est directement lié au nombre d'éléments à traiter. Avec un support si faible, un nombre très conséquent d'éléments est traité, et donc la différence entre les algorithmes se fait ressentir.

La dernière mesure que nous avons effectuée est présentée par la figure 4.9. Cette dernière mesure le temps d'exécution des deux algorithmes. La différence n'est pas significative ; en effet, au maximum, *SOARuleGrowth* est plus lent que son prédécesseur de 7.8%. Cette différence s'explique à cause du pré-traitement supplémentaire requis par *SOARuleGrowth* par rapport à *RuleGrowth*.

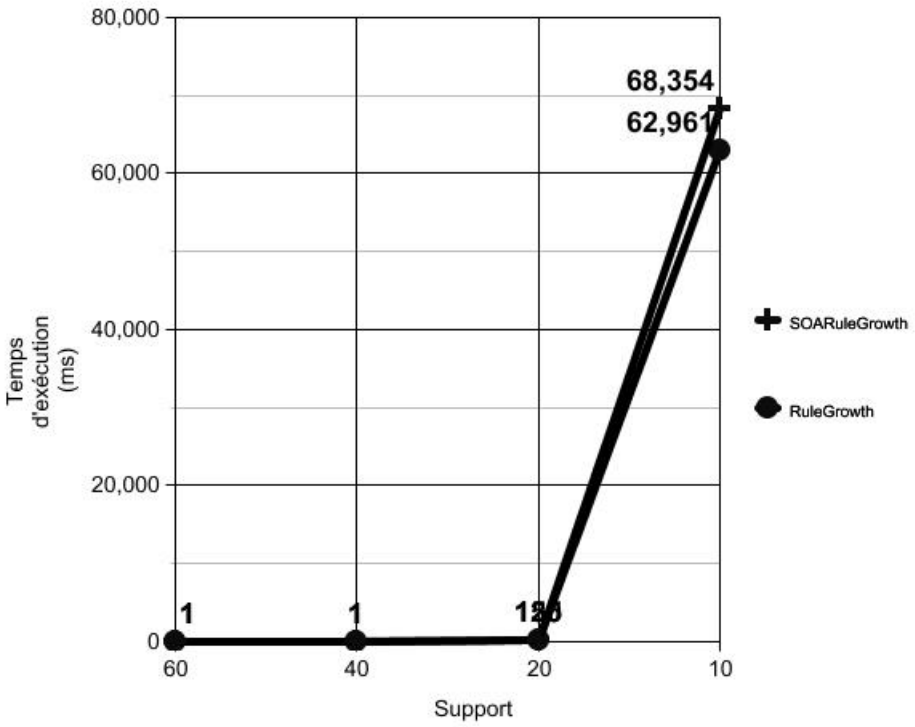


Figure 4.9: Temps d'exécution.

4.5 Changement d'objectif

Cette section explique pourquoi SOMAD obtient de meilleurs résultats que SODA pour identifier des anti-patrons qui sont plus à même d'endommager la qualité de service d'un SBS. En effet, la présence d'un anti-patron dans une partie peu visitée du système fait courir des risques plus faibles aux créateurs du système.

Les hypothèses de SOMAD changent l'objectif de la recherche d'anti-patrons SOA d'une considération de conception pure vers une considération d'utilisation. C'est à dire que nous cherchons à identifier les anti-patrons qui sont provoqués par la manière dont les utilisateurs consomment les services que le SBS offrent plutôt que par les collaborations statiques entre les services. De ce fait, SOMAD néglige les valeurs de métriques basiques. C'est un choix naturel, car SOMAD est une approche qui n'a pas accès aux valeurs exactes au travers des interfaces des services ou leurs implémentations. De plus, analyser un système à base de services via son utilisation plutôt que son architecture, comme l'indiquent les résultats expérimentaux décrits dans le chapitre 5, entraîne une précision bien supérieure. Considérons un service nommée `Half-Deprecated Service` (figure 4.10) composé de quatre méthodes : A, B, C and D. Les méthodes C et D sont dépréciées [3] mais sont tout de même exposées pour assurer la rétro-comptabilité de ce service avec ces clients.

Une méthode pour calculer la cohésion d'un service est de compter combien de méthodes du service sont utilisées durant une session unique d'un utilisateur unique. Comme la moitié des méthodes sont dépréciées, il est fort probable que le client ne consomme que la moitié des méthodes.

---

3. La dépréciation est, dans le domaine du développement logiciel, la situation où une ancienne fonctionnalité est considérée comme obsolète au regard d'un nouveau standard, bien qu'elle soit conservée dans les versions plus récentes pour des fins de compatibilité.

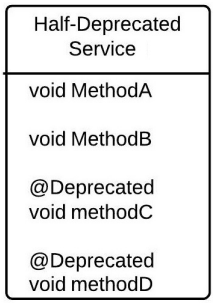


Figure 4.10: Représentation pseudo-UML du `Half-Deprecated Service`.

— Soit le client est un ancien client et il utilisera les deux méthodes dépréciées.
— Soit le client est un nouveau client et il utilisera les deux méthodes non dépréciées.

De ce fait, si la cohésion est calculée de cette manière, le résultat sera 0.5 (2/4) et ce service sera considéré comme suspect pour les anti-patrons SOA qui sont identifiables par une faible cohésion. Au contraire, si la cohésion est calculée en utilisant une méthode basée sur les traces d'exécution, le résultat aura plus de chance d'être près de 1. En effet, les appels aux méthodes dépréciées ne devraient pas apparaître dans les règles d'association séquentielles, car ils ne devraient pas atteindre le seuil minimum de confiance et de support.

4.6 Conclusion

Dans ce chapitre, nous avons présenté l'implémentation qui supporte l'approche SOMAD ainsi qu'une amélioration de *RuleGrowth* nommée *SOARuleGrowth* et des considérations sur le changement d'objectif dans la détection d'anti-patrons. Toutes ces modifications ont fait progresser notre précision de 3% par rapport à l'algorithme classique. De plus, le temps d'exécution de SOMAD (*SOARuleGrowth* + calcul des métriques et règles) est directement impacté par :

- *La taille des traces d'exécution.* En effet, plus le nombre de traces augmente, plus les algorithmes de pré-traitement et de génération de règles d'association vont avoir besoin de temps.
- *La complexité de connexion inter-services* (profondeur maximale des transactions). Ceci est dû au fait que le temps nécessaire au découpage des traces d'exécution en transaction est impacté par la profondeur de la pile.
- *Le nombre de métriques à calculer.* Une fois les règles d'association calculées il faut calculer les métriques des tableaux 3.1 et 3.2 afin de détecter les anti-patrons.
- *La complexité des règles de détection des anti-patrons* (nombres d'intersections, exclusions,...).
- *Le nombre de services.* En effet, les métriques doivent être calculées pour chaque service ; ainsi le nombre de services est un facteur important.

Dans le prochain chapitre, nous aurons un aperçu de la réaction de *SOARuleGrowth* face à la mise à l'échelle (*scalability*). En effet, nous avons mené nos expériences sur deux systèmes différents qui ont la particularité d'être composés de 13 et 130 services. Les traces d'exécutions produites par ces systèmes, quant à elles, passent de ∼1500 à ∼10 000 lignes. Malgré l'augmentation de la taille du système (x10) et celle des traces (x6.6), le temps d'exécution moyen par anti-patron n'est multiplié que par ∼4 passant de 0.068s à 0.280s. Nous pouvons donc dire que SOMAD supporte bien la mise à l'échelle ; d'autant plus qu'un système à base de services composé de 130 services peut être considéré comme un système de taille industrielle. Le chapitre 5 présentera aussi la validation empirique de notre approche et de nos algorithmes.

CHAPITRE V

EXPÉRIMENTATIONS ET VALIDATION

Afin de valider notre approche, nous avons appliqué SOMAD sur deux systèmes à base de services développés indépendamment, *HomeAutomation* et *FraSCAti* (Seinturier *et al.*, 2012). *HomeAutomation* est composé de 13 services tandis que FraSCAti est 10 fois plus important : 91 composants et 130 services. Nous avons choisi d'expérimenter notre approche sur ces systèmes car ce sont ceux qui ont permis de valider SODA — l'approche préliminaire — et nous allons de ce fait, pouvoir comparer les deux approches en termes de précision et de rappel d'un coté, et d'efficacité de l'autre.

Dans ce chapitre nous présenterons les hypothèses qui nous ont servi à valider nos expérimentations, puis nous présenterons en détail nos sujets d'expérimentation. Enfin, nous présenterons notre mode opératoire ainsi que nos résultats.

5.1 Hypothèses

Les expérimentations visent à valider les trois hypothèses suivantes :

Hypothèse 5.1 ***Précision.*** *Les algorithmes de détection doivent avoir un rappel de 100%, c'est-à-dire, que tous les anti-patrons présents sont détectés, et une précision supérieure à 75%, c'est-à-dire, que parmi les anti-patrons détectés, plus*

des trois-quarts sont de vrais positifs.

Cette première hypothèse supporte l'exactitude des règles d'association séquentielles générées ainsi que leur interprétation via nos métriques. De plus, cette hypothèse est similaire à celle qui ont permis de valider SODA car nous souhaitons que SOMAD soit, au minimum, aussi précis que SODA.

Hypothèse 5.2 ***Performance.*** *Le temps d'exécution requis par les algorithmes de détection sont aussi bas que ceux de SODA, c'est-à-dire, en dessous d'une seconde.*

Cette deuxième hypothèse supporte la performance de SOMAD par rapport à SODA en termes de temps d'exécution. Ce choix d'une seconde est une approximation grossière de ce qui peut être considéré comme non-intrusif en termes de temps d'attente pour un utilisateur souhaitant améliorer la qualité de son système à base de services via la détection d'anti-patrons. Dans un futur proche, nous souhaiterions pouvoir intégrer la détection d'anti-patrons pendant le développement des applications en analysant les traces produites à chaque compilation & lancement. De ce fait, 1 seconde nous paraît être acceptable.

Hypothèse 5.3 ***Extensibilité.*** *SOMAD est extensible dans le sens où on peut lui ajouter de nouveaux anti-patrons et les détecter.*

Avec cette dernière hypothèse à valider par nos expérimentations, nous voulons montrer combien il est aisé d'ajouter ou de combiner des métriques afin de détecter de nouveaux anti-patrons.

5.2 Sujets

Nous avons appliqué SOMAD pour détecter six anti-patrons SOA décrits dans le tableau 1.1. Dans la description de chaque anti-patron, nous avons mis en gras les caractéristiques importantes pour leurs détections via nos métriques des tableaux 3.1 et 3.2.

5.3 Objets

Une première passe d'expérimentation a été réalisée sur *HomeAutomation*. *HomeAutomation* est une application de type SCA développée indépendamment pour le contrôle domotique de maisons de personnes âgées (température, instruments électriques, urgences médicales). Cette application inclut 7 scénarios prédéfinis à des fins de tests et de démonstration. Deux versions différentes du système ont été utilisées : la version originale avec 13 services et une version dégradée intentionnellement dans laquelle des services ont été modifiés et d'autres ajoutés afin d'injecter de nouveaux anti-patrons SOA. Ces changements ont été réalisés par une tierce partie afin d'éviter de biaiser les résultats. La figure 5.1 présente l'architecture d'*HomeAutomation* tandis que la figure 5.2 présente son interface graphique.

Étant donné le manque de systèmes à base de services disponibles gratuitement, la seconde passe d'expérimentation a été réalisée sur le support d'exécution d'*HomeAutomation* : *FraSCAti* (Seinturier *et al.*, 2012). *FraSCAti* est aussi un système de type SCA composé de 91 composants et 130 services. Les services sont distribués dans les composants et un composant expose au moins un service. Contrairement à *HomeAutomation*, *FraSCAti* ne possède pas de scénarios prédéfinis — en réalité, il comporte quelques tests unitaires, mais pas de couverture complète des fonctionnalités. La détection a été effectuée en instrumentalisant *FraSCAti* afin qu'il produise des traces d'exécution tel que décrit dans le chapitre précedent. Comme

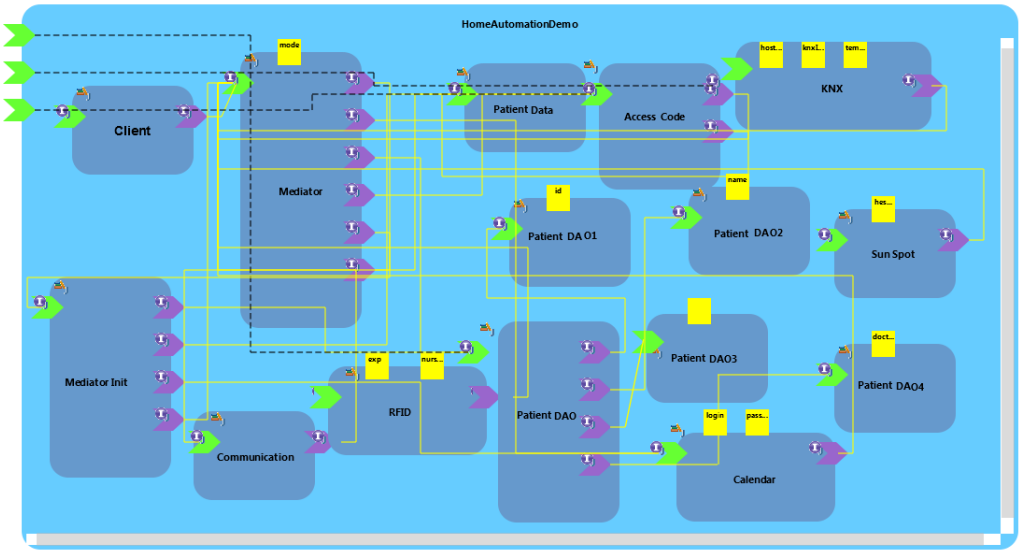


Figure 5.1: Diagramme SCA d'*HomeAutomation*.

Figure 5.2: Interface graphique d'*HomeAutomation*.

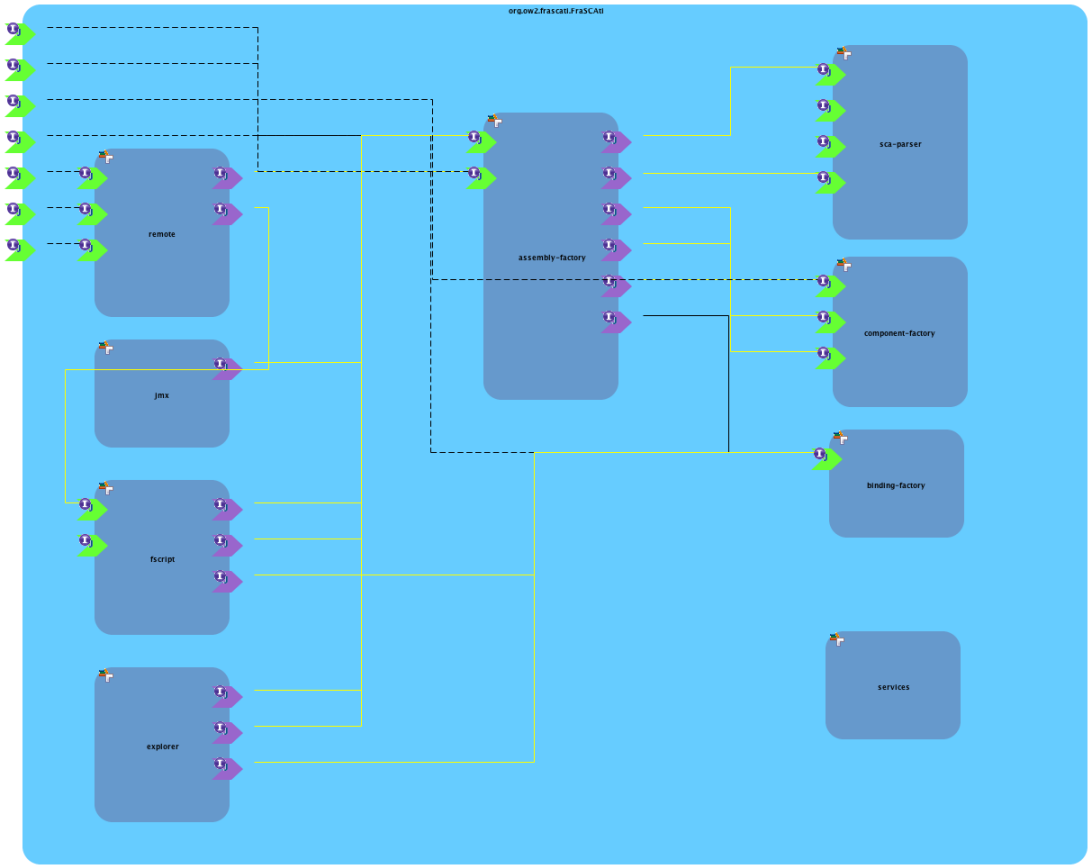


Figure 5.3: Diagramme SCA principal de *FraSCAti*.

FraSCAti est un environnement d'exécution pour les systèmes SOA, nous avons chargé et lancé des systèmes de diverses technologies (SCA, REST, Web-Service, RMI) dans *FraSCAti* ; puis nous avons utilisé ces systèmes dans le but de couvrir un maximum de fonctionnalités de *FraSCAti*.

La figure 5.3 présente l'architecture de FraSCAti tandis que la figure 5.4 présente l'interface graphique de l'explorateur de service.

La détection d'anti-patrons SOA sur *FraSCAti* a été effectuée au niveau des composants plutôt qu'à celui des services à cause de sa documentation. En effet,

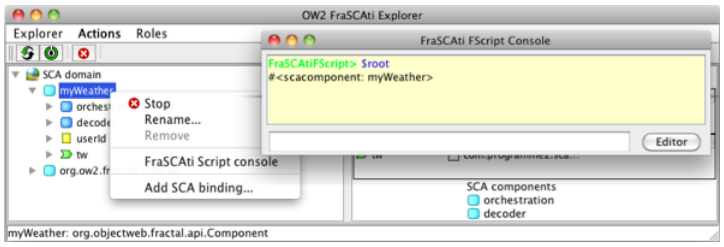


Figure 5.4: Interface graphique de l'explorer *FraSCAti*.

FraSCAti est documenté au niveau des composants et c'est cette documentation qui sera utilisée pour la validation des résultats. De plus, nous avons empiriquement prouvé que les systèmes SCA souffrent des mêmes maux architecturaux que les systèmes SOA purs.

D'autres détails sur les systèmes analysés peuvent être trouvés en ligne :

— `http://sofa.uqam.ca/somad`

— `http://frascati.ow2.org/doc/1.4/ch12s04.html`

Le tableau 5.1 présente un comparatif entre *HomeAutomation* et *FraSCAti* en termes de taille, nombre de services, nombre de méthodes et nombre de classes.

Application	Version	Taille	NDS	NDM	NDC
Home Automation	original	3.2 MLDC	13	226	48
Home Automation	dégradée	3.4 MLDC	16	243	52
FraSCAti	original	26.75 MLDC	130	1882	403

Tableau 5.1: Propriétés d' *Home-Automation* et *FraSCAti* (NDS : Nombre De Services, NDM : Nombre de Méthodes, NDC : Nombre de classe, MLDC : Milliers de lignes de code).

5.4 Matériel et langage

Les expérimentations ont été menées sur un poste avec les caractéristiques suivantes :

- Ubuntu Release 12.04 (precise) 64-bit
- Kernel Linux 3.5.0-36-generic
- Mémoire vive : 5.8 GB
- Processeur : 2x Intel Xeon(R) CPU E5345 @ 2.33GHz x 4

Le langage de programmation choisi pour l'implémentation de SOMAD est Java :

- Version : 1.6.0_27.
- Machine Virtuelle : OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode).

5.5 L'outil SOMAD

Nous avons développé l'outil SOMAD en adéquation avec l'approche du même nom. SOMAD a quatre fonctionnalités principales :

- La spécification d'anti-patrons SOA en utilisant les métriques de notre catalogue
- La détection automatique d'antipatrons dans les SOAs
- La visualisation du système cible

La figure 5.5 présente l'interface graphique de SOMAD. On y constate une visualisation des services présents sous forme de noeuds ainsi que les communications entre les services sous forme d'arêtes. Les services (noeuds) colorés en rouge signifient que le service est impliqué dans un anti-patron. L'outil affiche aussi la pile d'appel sur le panneau de gauche.

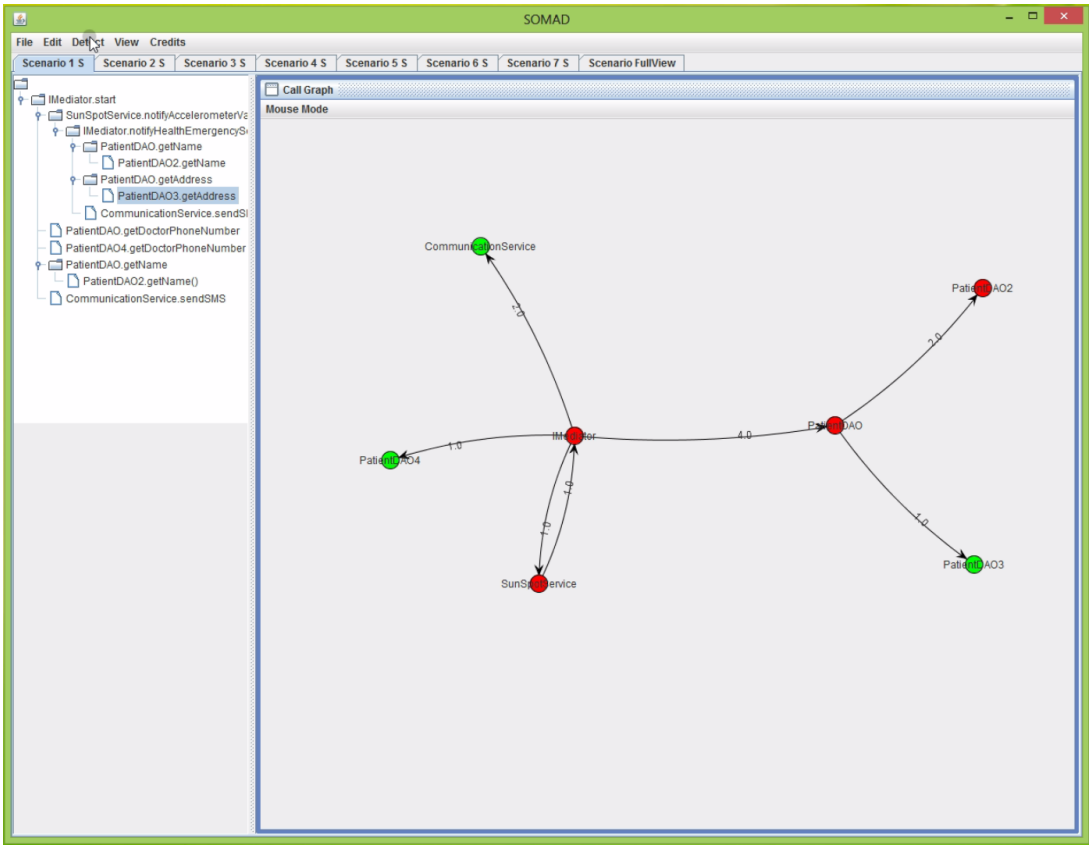


Figure 5.5: Interface de l'outil SOMAD.

De plus une autre fonctionnalité prometteuse est actuellement en cours de développement. Cette fonctionnalité consiste à ajouter de nouveaux services en utilisant leur définition (comme, par exemple, un WSDL), puis nous recalculons la détection des anti-patrons SOA en considérant ces ajouts. Une vidéo de l'outil SOMAD (en anglais) est disponible sur notre site institutionnel : `http://sofa.uqam.ca/somad/`.

5.6 Processus

Nous avons appliqué SOMAD pour la détection de six anti-patrons SOA sur deux systèmes distincts. Tout d'abord, nous avons lancé les sept scénarios d'*Home-Automation* :

- Détection de la chute d'un patient et envoi d'un message d'urgence au médecin contenant l'adresse du patient (Services impliqués : `SunSpot Service`, `Communication Service`, `Mediator`).
- Identique au scénario 1 mais planifie aussi un rendez-vous entre le patient et le médecin à une date ultérieure (Services impliqués : `SunSpot Service`, `Communication Service`, `Calendar Service`, `Mediator`).
- Ajout d'un rendez-vous dans l'agenda du docteur (Services impliqués : `Communication Service`, `Calendar Service`, `Mediator`).
- Contrôle des fenêtres et transfert de la responsabilité du patient à une infirmière ou un technicien (Services impliqués : `RFID Service`, `Mediator`).
- Contrôle des lumières dans la maison (Services impliqués : `RFID Service`, `Mediator`, `Communication Service`).
- Contrôle des lumières et des fenêtres (Services impliqués : `KNX Service`, `Mediator`).

Ensuite, nous avons chargé, lancé et utilisé cinq systèmes différents dans *FraSCAti* :

— Une calculatrice basée sur des services web.
— Un chat utilisant le Java RMI [1].
— Une implémentation de la suite de Fibonacci basée sur des services REST
— Une application de vente utilisant trois composants SCA dont un écrit en BPEL [2]
— Une application de météo consommant des services web distants

Une fois les traces générées, nous les avons réunies au sein d'un fichier distinct par application et extrait les transactions. Sur ces transactions, nous avons appliqué les algorithmes de fouille de règles d'association séquentielles avec un support minimum de 40% et une confiance minimum de 60%. Ces choix ne suivent pas d'indications spécifiques liées à l'ARM (*Association Rule Mining*) ou intuitions particulières sur nos sujets d'expérimentations. En effet, nous étions seulement guidés par le besoin de filtrer toutes les règles non pertinentes tout en gardant assez de règles pour représenter la majorité des appels. De plus, nous avions besoin d'un minimum de confiance élevée pour faire apparaître les alternatives les plus probables (conséquent) pour chaque conclusion de transaction (antécédent) tout en supprimant les moins significatives. De ce fait, nous avons réalisé plusieurs essais pour les deux seuils et observé la taille de l'ensemble de règles. Afin de déterminer quels étaient les meilleurs seuils, nous avons réalisé des essais incrémentaux en partant de 10% et 40%, respectivement pour le support et la confiance. Pour

1. *Remote Method Invocation*, plus connu sous l'acronyme RMI est une interface de programmation (API) pour le langage Java qui permet d'appeler des méthodes distantes

2. En informatique, Business Process Execution Language, est un langage de programmation destiné à l'exécution des processus d'affaires.

chaque essai, nous avons augmenté la valeur d'une des variables de 5% et observé la taille du résultat. Les valeurs actuelles semblent apporter le meilleur compromis entre la taille de l'ensemble de règles et la pertinence des règles.

L'étape suivante consiste en l'interprétation des règles d'association séquentielles générées. Dans ce but, nous avons appliqué nos métriques qui correspondent aux hypothèses basées sur les descriptions textuelles des anti-patrons. Enfin, nous avons validé les résultats en termes de précision et de rappel en analysant manuellement les systèmes. La précision (Equation 5.1) estime le ratio de vrais positifs dans les services suspects. Le rappel (Equation 5.2), quant à lui, estime le ratio d'anti-patrons détectés sur le total d'anti-patrons présents dans l'application.

$$précision = \frac{|\{anti\text{-}patrons\ existants\} \cap \{anti\text{-}patrons\ détectés\}|}{|\{anti\text{-}patrons\ détectés\}|} \tag{5.1}$$

$$rappel = \frac{|\{anti\text{-}patrons\ existants\} \cap \{anti\text{-}patrons\ détectés\}|}{|\{anti\text{-}patrons\ existants\}|} \tag{5.2}$$

Nous avons aussi évalué nos performances grâce à la mesure F_1 (Equation 5.3), qui est une moyenne pondérée de la précision et du rappel pour mesurer l'exactitude de nos algorithmes de détection.

$$Mesure\ F_1 = 2 \times \frac{précision \times rappel}{précision + rappel} \tag{5.3}$$

Cette validation a été réalisée manuellement par deux ingénieurs logiciels indépendants, à qui nous avons fourni les descriptions des anti-patrons, les deux versions d'*HomeAutomation* et le détail des composants de *FraSCAti*. Pour les deux systèmes, les résultats ont été comparés à ceux de SODA. Pour *FraSCAti*, nous avons rapporté notre détection à l'équipe en charge de son développement et obtenu une

validation objective de leur part.

5.7 Résultats

Le tableau 5.2 présente les résultats de la détection des six anti-patrons SOA sur *HomeAutomation*. Pour chaque anti-patron, le tableau contient les services détectés automatiquement par SOMAD, les services identifiés manuellement, les valeurs des métriques, le rappel et la précision, le temps de détection, et finalement la mesure F_1. De la même manière, le tableau 5.3 rapporte les résultats de la détection sur *FraSCAti*. Nous rappelons que les valeurs des métriques ne reflètent pas les valeurs réelles des services (par exemple, le nombre de méthodes), mais une représentation de la façon dont le système est utilisé (via les traces d'exécution). De plus, les valeurs des métriques sont pondérées par la fraction $\frac{support}{confiance}$ dans le but d'accentuer le poids des règles d'association séquentielles qui disposent de la plus grande confiance. De ce fait, un nombre de méthodes égal à 2 veut dire que parmi les règles d'association séquentielles générées, il y a 2 méthodes qui apparaîssent dans des règles disposant d'un fort support et d'une forte confiance.

Le temps requis moyen est de 174ms pour SOMAD et de 469ms pour SODA. Ceci est dû principalement au fait que SODA utilise la programmation orientée aspects (AOP, Aspect Oriented Programming) pour attacher le code des métriques sur chaque méthode de chaque service découvert. De plus, les métriques sont attachées une par une, par conséquent, l'exécution du SOA sous analyse est interrompue autant de fois qu'il y a de métriques à calculer, et ce à chaque invocation de méthodes. Au contraire, SOMAD n'interrompt pas l'application et n'est pas basé sur l'AOP, ainsi il offre de bien meilleures performances.

Anti-patrons	Services détectés automatiquement		Services détectés manuellement	Métriques SOMAD	Rappel	Précision	Temps	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%	[2/2] 100%	0.076s	100%
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, SunSpqtService, PatientDAO2}			[3/3] 100%	[3/4] 75%	0.056s	85.7%
Moyennes	SODA				100%	87.5%	0.231s	92.0%
	SOMAD				100%	95.8%	0.068s	97.6%

Tableau 5.2: Comparaison des résultats de SOMAD et SODA sur *HomeAutomation*. Les services barrés indiquent des faux-positifs détectés par *RuleGrowth* et pas par *SOARuleGrowth*.

Anti-patrons	Services détectés automatiquement		Services détectés manuellement	Métriques SOMAD	Rappel	Précision	Temps	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] N.A	[0/0] N.A	0.97s	N.A
	SOMAD	not present		NDP $\geqslant$ 24	[0/0] N.A	[0/0] N.A	0.77s	N.A
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 Metamdl-Provider	SCA-Composite	OC $\geqslant$ 3	[2/2] 100%	[2/3] 66.67%	0.076s	80%
Chain Service	SODA	SCA-Parser Composite-Mngr Processor	Composite Parser Composite-Mngr	LC $\geqslant$ 5	[2/2] 100%	[2/3] 66.67%	0.758	80%
	SOMAD	Composite-Parser Composite-Mngr			[2/2] 100%	[2/2] 100%	0.056s	100%
Moyennes	SODA				100%	73.33%	0.707s	84.62%
	SOMAD				100%	93.33%	0.28s	96.55%

Tableau 5.3: Comparaison des résultats de SOMAD et SODA sur *FraSCAti*.

5.8 Détails des résultats

Nous présentons les résultats de détection de SOMAD tout en les comparants à SODA, sur *HomeAutomation* et *FraSCAti*. Les résultats sont similaires, mis à part pour le *Knot* et le *BottleNeck*.

5.8.1 *HomeAutomation*

`IMediator` a été détecté et identifié comme un *Multi Service* par SODA et SOMAD, à cause de son grand nombre de méthodes (***N**umber of **M**ethods* ≥ 2), son grand nombre d'apparitions dans les règles (***N**umber of **MA**tches* ≥ 3.8) et sa faible cohésion (***COH*** ≤ 0.5). Les valeurs de ces métriques on été évaluées hautes et faibles en comparaison avec les scores obtenus par les autres services d'*HomeAutomation*. En effet, la technique de la boite à moustache (Boxplot en anglais) estime qu'un score ≥ 2 pour la métrique **NM** est fort dans la distribution statistique des valeurs obtenues par les autres services. De la même manière, les services détectés comme *Tiny Service* ont un petit nombre de méthodes (**NM** ≤ 2) et un fort couplage sortant (***O**utgoing **C**oupling* ≥ 4), à nouveau en accord avec la boite à moustache. Dans la version originale d'*HomeAutmation*, nous n'avons pas détecté de *Tiny Service*. Dans le but d'éprouver nos algorithmes de detection, un ingénieur indépendant a injecter cet anti-patrons. En effet, il a extrait une méthode du service `IMediator` et l'a déplacée dans un service nommé `MediatorDelegate` ; ce nouveau service a été détecté comme *Tiny Service*. Deux occurrences du *Chatty Service* ont été découvertes dans *HomeAutomation* par SODA et SOMAD. `PatienDAO` et `IMediator` apparaissent un grand nombre de fois dans les règles (**NMA** ≥ 3.8) — ce qui signifie qu'ils communiquent avec beaucoup d'autres services — et ils ont un grand nombre de partenaires différents (**N**umber of **D**ifferent **P**artners ≥ 0.6).

`PatientDAO` a été détecté comme un *Knot* car il une forte dépendance cyclique d'invocation (***C**ross **I**nvocation **D**ependencies* ≥ 2). Dans *HomeAutomation*, un ensemble de services `PatientDAO1`, `PatientDAO2`, `PatientDAO3` et `PatientDAO4` sont fortement couplés car chacun d'entre eux représente une partie des informations d'un patient (nom, adresse, numéro de téléphone, numéro de téléphone du docteur). De ce fait, des invocations cycliques systématiques entre ces services apparaissent quand le système souhaite accéder aux informations complètes du patient. SOMAD n'a pas détecté le faux positif `IMediator` détecté par SODA, et de ce fait, obtient une meilleure précision pour cet anti-patron.

Deux services ont été détectés comme étant des *BottleNeck* : `IMediator` et `PatientDAO` à cause de leur fort couplage entrant ($\mathbf{OC} \geq 3$) et sortant (***I**ncoming* - ***C**oupling* ≥ 4).

Finalement, SODA et SOMAD ont détecté une chaîne d'invocations transitives ou *Service Chain* : `IMediator` $\rightarrow$ `PatientDAO` $\rightarrow$ `PatientDAO2` $\rightarrow$ `SunSpotService` ($\mathbf{TC} \geq 4$). Les deux approches rapportent le faux positif `SunSpotService`.

5.8.2 *FraSCAti*

Nous présentons maintenant les résultats des détections effectuées sur *FraSCAti*.

`SCA-Parser` est suspecté d'être un *Tiny Service* car il a un faible nombre de méthodes ($\mathbf{NM} \leq 1$) et un fort couplage sortant ($\mathbf{OC} \geq 3$). Une inspection manuelle du code de *FraSCAti* a révélé que le service `SCA-Parser` ne contient qu'une seule méthode nommée `parse`. L'équipe de développement de *FraSCAti* a validé cette détection. Ils ont indiqué que ce service peut être utilisé seul uniquement quand la lecture d'un fichier SCA est demandée. Cependant, *FraSCAti* permet d'effectuer un grand nombre de tâches qui auront besoin de `SCA-Parser`. Ces autres tâches sont déléguées à d'autres services tels que `AssemblyFactory`. Ceci explique le fort

couplage sortant.

SOMAD n'a pas détecté de *Multi Service* dans *FraSCAti*. Cependant, l'inspection manuelle a révélé que le composant `Explorer-GUI` en est une occurence. L'équipe de développement de *FraSCAti* a validé que ce composant utilise un grand nombre d'autres services fournis par d'autres composants. En effet, ce composant encapsule l'interface graphique de l'explorateur *FraSCAti* qui fournit une interface exhaustive de toutes les fonctionnalités offertes par *FraSCAti*. SOMAD n'a pas été capable de le détecter car le processus d'expérimentation n'implique, à aucun moment, l'interface graphique.

SOMAD n'a pas détecté de *Chatty Service*. En effet, aucun service ne dispose de nombreuses apparitions dans les règles (**NMA**) et d'un grand nombre de partenaires différents (**NDP**), respectivement supérieurs à 70 et 24 qui représentent les seuils minimaux pour être considéré comme haut par la boite à moustache. Cela signifie qu'aucun service n'apparait plus de 70 fois dans les règles et ne communique avec plus de 24 autres services. L'inspection manuelle de *FraSCAti* n'a pas permis d'identifier un tel anti-patron. Nous avons exclus les résultats pour le *chatty service* du calcul des moyennes. En effet, nous savons que SOMAD ne détecte pas de faux positifs sur cet anti-patron, mais nous ne pouvons pas être sûr que SOMAD détecterai un vrai positif s'il en rencontrai un dans *FraSCAti*.

Le composant `Metamodel-Provider` est suspecté de faire partie d'un *Knot* car il a une faible cohésion (**COH** ≤ 0.2) et un nombre important de dépendances cycliques (**CID** ≥ 25). La validation par l'équipe de développement de *FraSCAti* confirme que ce composant est au centre de large flux de communications multidirectionnelles, mais ils ne sont pas d'accord sur la spécification de cet anti-patron particulier.

SOMAD a aussi détecté trois occurrences de *BottleNeck* : `SCA-Parser`, `Composite-`

`Parser` et `Metamodel-provider`. Toutefois, le dernier des trois est un faux positif. Ces services ont été identifiés comme *BottleNeck* car ils disposent d'un fort couplage sortant et entrant, tous deux supérieur ou égal à 3.

Finalement, le `Composite-Parser` ($\mathbf{TC} \geq 4$) a été détecté et identifié comme partie d'un *Service Chain*, tandis que le `Composite~Manager` est un faux positif. L'équipe de développement de *FraSCAti* a confirmé que le `Composite-Parser` utilisait une chaîne de délégation pour compléter son abstraction.

Nous pouvons observer que le `Composite-Parser` et `SCA-Parser` sont des services suspects. Ces services sont très couplés avec d'autres services et, en particulier, ils font partie de longues chaînes d'invocations. La présence de tels anti-patrons dans ce système s'explique car il n'existe pas de manière de développer de parseur sans introduire un fort couplage et une forte transitivité.

En conclusion, *FraSCAti* obtient de bons résultats quant à la détection d'anti-patrons. Peu de services ont été détectés comme tels en comparaison du grand nombre de services/composants présents dans le système.

5.8.3 Etude des faux positifs

Nous étudions maintenant les raisons qui ont menés à la détection de faux positifs dans *HomeAutomation* et *FraSCAti*.

HomeAutomation

Le seul faux positif dans *HomeAutomation* concerne le `SunSpotService` en tant que *Chain Service*. Étant donné que SODA détecte aussi ce faux positif, et après vérification des analyses manuelles, il semblerait que `SunSpotService` ne soit pas un faux positif, mais une erreur apportée par l'analyse manuelle.

FraSCAti

Le seul faux positif détecté par SOMAD sur *FraSCAti* est le `Metamodel-Provider` en tant que *BottleNeck*. Une caractéristique non négligeable pour identifier un *BottlNeck* est son fort temps de réponse dû au fait qu'il est un goulot d'étranglement dans le système. Néanmoins, nous avons déployé tous les services de *FraSCAti* en local lors de nos expérimentations et n'avions donc aucune information pertinente à tirer des temps de réponses des services — compris entre 0.1s et 0.5s. Afin d'augmenter notre précision sur cet anti-patron, nous devrions considérer l'ajout d'une nouvelle métrique **RT** (*Repsonse Time*) qui mesurerait le temps de réponse des services en utilisant les marqueurs temps des traces d'exécution. Cependant, cette métrique s'appliquerait directement sur les traces, et non sur les règles d'association, ce qui pose un problème de compatibilité d'informations. En effet, nous ne pourrions que difficilement associer un temps d'exécution précis à une partie de règle : il nous faudrait utiliser des moyennes, ce qui est sans aucun doute discutable.

5.9 Discussion sur les hypothèses

Nous allons maintenant vérifier chacune de nos trois hypothèses posées précédemment en utilisant les résultats de la détection.

Hypothèse 5.4 ***Précision.*** *Les algorithmes de détection doivent avoir un rappel de 100%, c'est-à-dire, que tous les anti-patrons présents sont détectés, et une précision supérieure à 75%, c'est-à-dire, que parmi les anti-patrons détectés, plus des trois-quarts sont de vrai positifs.*

Comme indiqué dans les tableaux 5.2 et 5.3, nous avons obtenu un rappel de 100%, ce qui veut dire que tous les anti-patrons existants ont été détectés. La

précision, quant à elle, est de 90.1% pour *Home-Automation* et de 93.3% pour *FraSCAti*. La précision est supérieure de 8.3% à 20% par rapport à SODA. Des écarts importants en faveur de SOMAD sont aussi à signaler au niveau de la mesure F_1. En effet, SOMAD a obtenu des valeurs de 97.6% et 96.6% pour *Home-Automation* et *FraSCAti*, respectivement. Ces valeurs sont supérieures à celle obtenues par SODA, d'une marge allant de 5.6% à 11.9%.

Nous validons donc notre première hypothèse. De plus, SODA n'est capable d'analyser que des systèmes SCA alors que SOMAD peut être utilisé sur des traces d'exécution provenant de toutes les technologies d'implémentation SOA.

Hypothèse 5.5 ***Performance.*** *Le temps d'exécution requis par les algorithmes de détection sont aussi bas que ceux de SODA, c'est-à-dire, en dessous d'une seconde.*

Les expérimentations ont été menées 10 fois, et nous avons reporté la moyenne des temps d'exécution pour chaque anti-patron. Pour tous les anti-patrons, les temps de détection sont largement en dessous de la seconde, et ce, quel que soit le système. En effet, ils sont compris entre 0.05s et 0.70s. La moyenne des temps d'exécution est de 0.17s pour SOMAD soit, 2.5 fois plus rapide que SODA. Cependant, dans SOMAD, les temps d'exécution comprennent l'analyse des traces d'exécution. De ce fait, le temps requis augmentera considérablement avec l'augmentation du nombre de traces d'exécution à analyser. En effet, plus de 80% du temps actuel est dédié à la reconstruction des transactions et à la génération des règles d'association séquentielles. De plus, l'augmentation du temps d'exécution n'est pas linéaire : le facteur entre *Home-Automation* et *FraSCAti* est de 10, néanmoins, le temps d'exécution requis pour *FraSCAti* est 12 fois supérieur à celui requis pour *HomeAutomation*. Cependant, un système à base de services composé

de 91 composants et 130 services est un système de grande ampleur, et les temps de détection sont inférieurs à 1s. Nous validons donc notre seconde hypothèse.

Hypothèse 5.6 ***Extensibilité.*** *SOMAD est extensible dans le sens où on peut lui ajouter de nouveaux anti-patrons et les détecter.*

La preuve de concept de SOMAD n'était constituée que de quatre métriques, basées sur deux hypothèses et visant la détection de trois anti-patrons. Cela nous a pris moins d'une demi-journée de travail pour ajouter chaque nouvelle métrique. Ainsi, nous validons notre troisième et dernière hypothèse.

5.10 Obstacles possibles à la validité

Le principal obstacle à la validation de nos résultats est la *validation externe*, c'est-à-dire, la possibilité de généraliser nos résultats actuels à toutes les autres technologies SOA. Étant donné le manque de systèmes disponibles (gratuitement), nous avons fait de notre mieux pour obtenir des systèmes de taille réelle tels que *FraSCAti* et nous avons mené nos expériences sur deux versions d'*HomeAutomation*. Cependant, nous prévoyons d'effectuer des expérimentations sur des traces provenant d'autres systèmes tels que REST ou services web.

Pour la *validation interne*, les résultats de nos détections dépendent de nos hypothèses. Celles-ci semblent êtres pertinentes et validées car nous obtenons des résultats similaires et meilleurs à ceux de SODA. La *validation interne* est donc maitrisée.

La nature subjective de l'interprétation des règles d'association séquentielles et de la validation des anti-patrons est une menace à la *validité de construction*. Nous avons contrôlé cette menace car nous avons spécifié nos hypothèses en nous aidant

de la littérature des anti-patrons et en impliquant deux ingénieurs indépendants ainsi que l'équipe de développement de *FraSCAti* dans notre étude.

Finalement, nous avons minimisé la menace de *fiabilité* en automatisant la génération des règles d'association séquentielles et les algorithmes de détection.

CONCLUSION

La détection d'anti-patrons SOA est une activité cruciale pour assurer la qualité de conception des systèmes à base de services. Dans ce mémoire, nous avons présenté une approche innovatrice nommée SOMAD pour la détection de tels anti-patrons. Cette approche repose sur deux techniques complémentaires, venant de deux champs de recherche très actifs dans l'ingénierie des logiciels : la fouille dans les traces d'exécution et la surveillance informatique de systèmes, tous deux appliqués dans un environnement SOA. Plus précisément, SOMAD détecte les anti-patrons SOA en fouillant des règles d'association séquentielles en utilisant une adaptation de *RuleGrowth* nommée *SOARuleGrowth*. Ensuite, SOMAD filtre la connaissance que les règles d'association séquentielles contiennent à propos des relations entre les services en utilisant une suite de métriques dédiées. L'utilité de SOMAD a été démontrée en l'appliquant à deux systèmes à base de services développés indépendamment. Les résultats de cette approche incrémentée depuis SODA – l'unique et par conséquent outil de l'état de l'art, afin de combler ses limitations – ont été comparés à ceux de SODA et montrent que SOMAD obtient de meilleurs résultats. Sa précision est meilleure par une marge allant de 8.3% à 20% tout en gardant le rappel à 100% et étant, au minimum, 2.5 fois plus rapide. De plus, SOMAD utilise les traces d'exécution comme données d'entrées, traces d'exécution qui peuvent provenir de n'importe quelles implémentations SOA. Au contraire, SODA était focalisé sur les systèmes à base de composants.

Pour l'avenir, nous imaginons SOMAD dans le contexte d'un centre de données où son but serait d'optimiser les communications de ce dernier. Nous devrions aussi

investiguer des techniques alternatives de génération de règles d'association séquentielles afin de raffiner notre approche avec de nouvelles informations, comme par exemple, la fouilles de graphe (Chakrabarti et Faloutsos, 2006) ou la détection de patrons récurrents d'anomalies comportementales en utilisant la fouille de patrons rares (Szathmary *et al.*, 2007). Finalement, nous pourrions combiner des représentations sémantiques, c'est-à-dire, des ontologies OWL [3], et des méthodes de fouille applicables aux graphes libellés hétérogènement (Adda *et al.*, 2010). En effet, ce domaine semble particulièrement pertinent et prometteur pour nos recherches.

3. *Web Ontology Language* (OWL) est un langage de représentation des connaissances construit sur le modèle de données de RDF. Il fournit les moyens pour définir des ontologies web structurées.

Appendices

APPENDICE I

IMPROVING SOA ANTIPATTERNS DETECTION IN SERVICE BASED SYSTEMS BY MINING EXECUTION TRACES

Publié dans les actes de la WCRE'13 (Working Conference on Reverse Engineering 2013) par **Mathieu Nayrolles**, Naouel Moha et Petko Valtchev.

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

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.

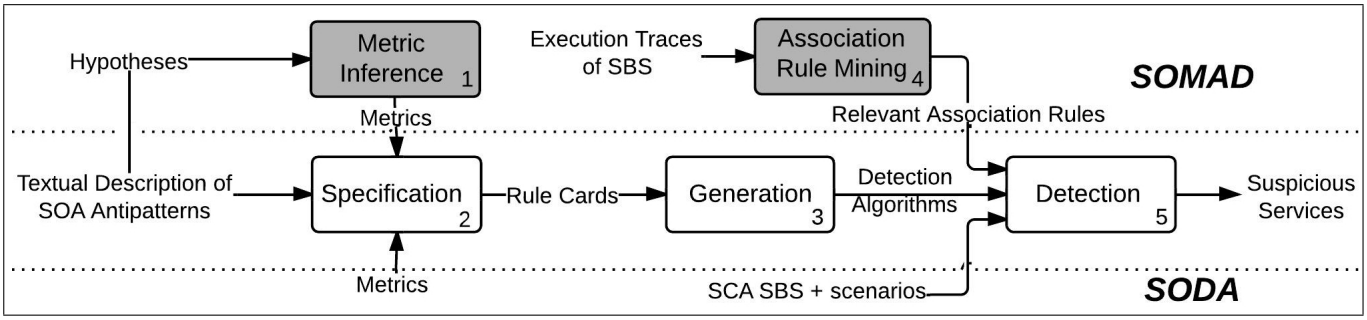


Fig. 1: SODA and SOMAD approaches: Grey boxes depict new steps in SOMAD w.r.t. to SODA (white boxes).

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 cohesive when all of its service operations are invoked by all the clients of this service*" [27]. Based on the above three

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

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.

TABLE I: List of SOA Antipatterns [2]

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

```
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

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 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

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.

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

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 a follow-up study, we shall be implementing a method specifically targeting traces. 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` because 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`

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 III: Results comparison between SODA & SOMAD on HomeAutomation

$\rightarrow$ 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 *Bottle-Neck 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

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%

TABLE IV: Results comparison between SODA & SOMAD on FraSCAti

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.

RÉFÉRENCES

Adda, M., Valtchev, P., Missaoui, R. et Djeraba, C. (2010). A framework for mining meaningful usage patterns within a semantically enhanced web portal. Dans *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, 138–147.

Agrawal, R. et Srikant, R. (1994). Fast Algorithms for Mining Association Rules. Dans *International Conference on Very Large Data Bases*, 1–32.

Asbagh, M. J. et Abolhassani, H. (2007). Web Service Usage Mining : Mining For Executable Sequences. Dans *International Conference on Applied Computer Science*, 266–271.

Brown, W., Malveau, R. et Mowbray, T. (1998). *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis*. Wiley.

Chakrabarti, D. et Faloutsos, C. (2006). Graph mining. *ACM Computing Surveys*, *38*(1), 1–68.

Cherbakov, L., IbrahimJ, M. et An, J. (2005). SOA antipatterns.

Daigneau, R. (2011). *Service Design Patterns : Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, volume 304. Addison-Wesley.

Demange, A., Moha, N. et Tremblay, G. (2013). Detection of SOA Patterns. In *International Conference on Service-Oriented Computing* 114–130.

Dijkman, R., Dumas, M., Garcia-Banuelos, L. et Kaarik, R. (2009). Aligning Business Process Models. Dans *2009 IEEE International Enterprise Distributed Object Computing Conference*, 45–53.

Dudney, B. (2003). *J2EE AntiPatterns*. John Wiley \& Sons.

Dustdar, S. et Gombotz, R. (2006). Discovering web service workflows using web services interaction mining.

Erl, T. (2008). *SOA : principles of service design*. Prentice Hall.

Erl, T. (2009). *SOA Design Patterns*. Prentice Hall.

Fokaefs, M., Tsantalis, N. et Chatzigeorgiou, A. (2007). JDeodorant : Identification and Removal of Feature Envy Bad Smells. Dans *International Conference on Software Maintenance*, 519–520.

Fournier, P.-m. et Dagenais, M. R. (2009). Combined Tracing of the Kernel and Applications with LTTng. Dans *Linux Symposium*, 209–224.

Fournier-Viger, P., Faghihi, U., Nkambou, R. et Nguifo, E. M. (2012). CMRules : Mining sequential rules common to several sequences. *Knowledge-Based Systems*, *25*(1), 63–76.

Fournier-Viger, P., Nkambou, R. et Tseng, V. S.-M. (2011). RuleGrowth. Dans *Proceedings of the 2011 ACM Symposium on Applied Computing*, p. 956., New York, New York, USA. ACM Press.

Fowler, M., Beck, K., Brant, J. et Opdyke, W. (1999). *Refactoring : Improving the Design of Existing Code*. Addison Wesley.

G. Piatetsky-Shapiro (1991). Discovery, analysis and presentation of strong rules. *Knowledge Discovery in Databases*, 229–249.

Hamilton, H. J. and Karimi, K. (2005). Advances in Knowledge Discovery and Data Mining. Dans T. B. Ho, D. Cheung, et H. Liu (dir.). *9th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 3518 de *Lecture Notes in Computer Science*, 744–750., Berlin, Heidelberg. Springer Berlin Heidelberg.

Hu, L. et Sartipi, K. (2008). Dynamic Analysis and Design Pattern Detection in Java Programs. Dans *Software Engineering & Knowledge Engineering*, 842–846.

Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M. et Ouni, A. (2011). Design Defects Detection and Correction by Example. Dans *International Conference on Program Comprehension*, 81–90. IEEE.

Kessentini, M., Vaucher, S. et Sahraoui, H. (2010). Deviance from perfection is a better criterion than closeness to evil when identifying risky code. Dans *International Conference on Automated Software Engineering*, 113–122. ACM Press.

Khan, A., Lodhi, A. et Veit, K. (2010). Applying Process Mining in SOA Environments. Dans *International Conference on Service Oriented Computing*, 293–302.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G. et Sahraoui, H. (2011). BDTEX : A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software. Elsevier*, *84*(4), 559–572.

Kral, J. et Zemlicka, M. (2009). Crucial Service-Oriented Antipatterns. *International Journal On Advances in Software*, *2*(1), 160–171.

Lanza, M. et Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Springer.

Liang, Q., Chung, J.-y., Miller, S. et Ouyang, Y. (2006). Service Pattern Discovery of Web Service Mining in Web Service Registry-Repository. Dans *International Conference on e-Business Engineering*, 286–293. IEEE.

Mannila, H., Toivonen, H. et Verkamo, A. I. (1997). Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, *1*(3), 259–289.

Marinescu, R. (2004). Detection strategies : metrics-based rules for detecting design flaws. Dans *International Conference on Software Maintenance*, 350–359. IEEE.

Michael S., M. et Jacob, J. (1971). Is there an optimal number of alternatives for Likert scale items ? I. Reliability and validity. *Educational and Psychological Measurement*, *31*(3), 657–674.

Moha, N., Gueheneuc, Y.-G., Duchien, L. et Le Meur, A.-F. (2010). DECOR : A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, *36*(1), 20–36.

Moha, N., Palma, F., Nayrolles, M., Joyen-Conseil, B., Guéhéneuc, Y.-G., Baudry, B. et Jézéquel, J.-M. (2012). Specification and Detection of SOA Antipatterns. Dans *International Conference on Service Oriented Computing*, 1–16. Springer.

Mortazavi-Asl, B., Pinto, H. et Dayal, U. (2004). Mining sequential patterns by pattern-growth : the PrefixSpan approach. *IEEE Transactions on Knowledge and Data Engineering*, *16*(11), 1424–1440.

Nayrolles, M., Moha, N. et Valtchev, P. (2013). Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces. Dans *Working Conference on Reverse Engineering*, numéro i, 321–330. IEEE.

Nayrolles, M., Palma, F., Moha, N. et Guéhéneuc, Y.-G. (2012). SODA : A Tool Support for the Detection of SOA Antipatterns. Dans *International Conference on Service Oriented Computing LNCS 7759*, 451–456. Springer.

Ng, J. K.-Y., Gueheneuc, Y.-G. et Antoniol, G. (2010). Identification of Behavioral and Creational Design Motifs through Dynamic Analysis. *Journal of Software Maintenance and Evolution : Research and Practice*, *22*, 597–627.

Obeo (2005). *Acceleo.* Rapport technique, Eclipse Fundation.

Palma, F. (2013). *Detection of SOA Antipatterns.* (Thèse de doctorat). Ecole Polytechnique de Montreal.

Palma, F., Nayrolles, M. et Moha, N. (2013). SOA Antipatterns : An Approach for their Specification and Detection. *International Journal of Cooperative Information Systems*, *22*(04), 1–40.

Parsons, T. (2007). *Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems.* (Thèse de doctorat). University College Dublin.

Pei, J., Han, J. et Mortazavi-asl, B. (2000). Mining Access Patterns Efficiently from Web Logs *. Dans *Knowledge Discovery and Data Mining. Current Issues and New Applications*, volume 0, 396–407.

Perepletchikov, M., Ryan, C., Frampton, K. et Tari, Z. (2007). Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. Dans *2007 Australian Software Engineering Conference*, 329–340. IEEE.

Perepletchikov, M., Ryan, C. et Tari, Z. (2010). The Impact of Service Cohesion on the Analyzability of Service-Oriented Software. *IEEE Transactions on Services Computing*, *3*(2), 89–103.

Rotem-Gal-Oz, Arnon Bruno, E. et Dahan, U. (2012). *SOA Patterns*. Manning.

Rutar, N., Almazan, C. et Foster, J. (2004). A Comparison of Bug Finding Tools for Java. Dans *15th International Symposium on Software Reliability Engineering*, 245–256. IEEE.

Safyallah, H. et Sartipi, K. (2006). Dynamic Analysis of Software Systems using Execution Pattern Mining. Dans *International Conference on Program Comprehension*, 84–88. IEEE.

Sciamma, D., Cannenterre, G. et Lescot, J. (2013). *Ecore Tools. Technical report, last update : May 2013.* Rapport technique.

Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V. et Stefani, J.-B. (2012). A component-based middleware platform for reconfigurable service-oriented architectures. *Software : Practice and Experience*, *42*(5), 559–583.

Settas, D. L., Meditskos, G., Stamelos, I. G. et Bassiliades, N. (2011). SPARSE : A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies. *Expert Systems with Applications*, *38*(6), 7633–7646.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A. et Katz, Y. (2007). Pellet : A practical OWL-DL reasoner. *Web Semantics : Science, Services and Agents on the World Wide Web*, *5*(2), 51–53.

Szathmary, L., Napoli, A. et Valtchev, P. (2007). Towards Rare Itemset Mining. Dans *19th IEEE International Conference on Tools with Artificial Intelligence*, 305–312. IEEE.

Tonella, P. et Ceccato, M. (2004). Aspect mining through the formal concept analysis of execution traces. Dans *11th Working Conference on Reverse Engineering*, 112–121. IEEE Comput. Soc.

Trčka, N., van der Aalst, W. M. P. et Sidorova, N. (2009). Data-Flow Anti-patterns : Discovering Data-Flow Errors in Workflows. Dans P. Eck, J. Gordijn, et R. Wieringa (dir.). *Advanced Information Systems Engineering*, volume 5565 de *Lecture Notes in Computer Science*, 425–439., Berlin, Heidelberg. Springer Berlin Heidelberg.

Upadhyaya, B., Tang, R. et Zou, Y. (2012). An approach for mining service composition patterns from execution logs. *IEEE Journal of Software : Evolution and Process*, *5*(3), 678– 679.

Weijters, A. et van der Aalst, W. (2003). Rediscovering workflow models from event-based data using little thumb - Volume 10, Number 2/2003 - IOS Press. *Integrated Computer-Aided Engineering*, *10*(10), 151–162.

Wilde, N., Simmons, S., Pressel, M. et Vandeville, J. (2008). Understanding features in SOA. Dans *2nd International workshop on Systems development in SOA environments*, 59–62., New York, New York, USA. ACM Press.

Wong, S., Aaron, M., Segall, J., Lynch, K. et Mancoridis, S. (2010). Reverse Engineering Utility Functions Using Genetic Programming to Detect Anomalous Behavior in Software. Dans *2010 17th Working Conference on Reverse Engineering*, 141–149. IEEE.

Yang, D.-L., Hsieh, Y. et Wu, J. (2006). Using Data Mining to Study Upstream and Downstream Causal Relationship in Stock Market. Dans *Proceedings of the 9th Joint Conference on Information Sciences*, 1–4., Paris, France. Atlantis Press.

Yousefi, A. et Sartipi, K. (2011). Identifying distributed features in SOA by mining dynamic call trees. Dans *Iinternational Conference on Software Maintenance*, 73–82. IEEE.