UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DÉTECTION DE PATRONS DE CONCEPTION DANS LES ARCHITECTURES ORIENTÉES SERVICES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ANTHONY DEMANGE

MARS 2014

REMERCIEMENTS

Mes plus sincères remerciements sont à l'attention de mes deux directeurs de recherche : Guy Tremblay, professeur et directeur du département d'informatique, et Naouel Moha, professeure au département d'informatique. Tous deux m'ont accordé une grande attention et une présence sans faille tout au long de ces deux années de maîtrise, et ceci malgré leur charge de travail importante. Je tiens aussi tout particulièrement à les remercier pour leur support financier, qui m'a notamment permis d'être récipiendaire de la bourse d'excellence, support sans lequel mes conditions de travail n'auraient pas été les mêmes.

Mes remerciements s'adressent aussi à mes collègues et amis du département d'informatique : Alexandre, Anis, Antonin, Benoît, Etienne, Geoffrey, Lucas, Mathias, Mathieu N., Mathieu S., et d'autres dont j'oublie certainement les noms. Merci à vous, mes amis, pour votre présence, pour votre bonne humeur, vous avez su égayer les longues journées de travail dans le laboratoire LATECE et me donner le courage nécessaire à l'accomplissement de mes travaux de recherche.

Je ne saurai assez remercier ma moitié, Peggy, qui a su être tolérante, patiente, attentionnée, et forte durant ces longs mois d'absence et de distance. Je ne serai jamais assez reconnaissant pour ton soutien, je sais combien l'épreuve a été longue et difficile mais nous savons tous deux que nos projets d'avenir n'en seront que plus enrichissants !

Sans oublier ma famille, pour qui la distance a clairement été difficile à concevoir et que je remercie infiniment pour sa compréhension. Je remercie mes parents de m'avoir permis de partir étudier, sans retenue, dans un autre pays et de m'avoir octroyé les ressources et le soutien nécessaire.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

On assiste de nos jours à une démocratisation des architectures orientées services (SOA : Service Oriented Architecture) afin de permettre le déploiement de systèmes d'information à grande échelle. Cependant, les systèmes basés sur ce type d'architecture évoluent rapidement du fait de l'ajout de nouvelles fonctionnalités ou l'intégration de systèmes légataires. La maintenance de ces systèmes devient donc de plus en plus complexe et implique alors de forts coûts de développement. Afin de faciliter l'évolution et la maintenance des systèmes SOA, ceux-ci doivent satisfaire de bons critères de conception et de qualité de service pouvant être exprimés via des patrons. Le terme patron fait référence à de bonnes pratiques permettant de résoudre des problèmes communs et récurrents de conception logicielle. Notre étude vise la spécification et la détection automatique de patrons SOA afin d'évaluer la qualité de systèmes SOA. Nous proposons l'approche SODOP (*Service Oriented Detection of Patterns*) basée sur une précédente approche pour la détection d'anti-patrons. Lors de la première étape, nous utilisons des cartes de règles afin de spécifier cinq patrons SOA extraits de notre revue de la littérature. Une carte de règles définit un ensemble de règles combinant plusieurs métriques, statiques ou dynamiques, à l'aide d'une grammaire formelle. Des algorithmes de détection correspondant à chaque carte de règles sont générés automatiquement durant la deuxième étape. La dernière étape consiste à concrètement appliquer ces algorithmes sur les systèmes SOA à analyser. Nous validons l'approche SODOP sur deux systèmes SOA : *Home-Automation* et *FraSCAti*, qui contiennent respectivement 13 et 91 services. Cette validation montre que l'approche proposée est précise, efficace et performante.

INTRODUCTION

L'architecture orientée services (SOA : *Software Oriented Architecture*) est un style architectural de plus en plus adopté de nos jours. En effet, ce style fournit aux architectes de systèmes une solution de haut niveau de conception de programmes informatiques. Les systèmes basés sur SOA sont conçus via de multiples composants autonomes, réutilisables, peu couplés et à gros grain que l'on appelle *services* (Rotem-Gal-Oz, 2012). Chaque service expose un comportement spécifique au domaine pour lequel il a été créé. Cependant, plusieurs services peuvent être combinés en un service composite pour répondre à des processus d'affaires de haut niveau d'exigence, qui nécessitent plusieurs opérations sous-jacentes. Différentes technologies existent actuellement pour implémenter ce style architectural. Parmi elles, les Services Web (Hansen, 2007) et les architectures orientées composants (SCA : *Service Component Architecture*) (Chappell, 2007) sont les plus communément utilisées. Google, Amazon et Microsoft font partie des sociétés les plus connues qui ont basé une partie ou tout leur système d'information sur l'approche SOA.

Les systèmes logiciels évoluent rapidement notamment grâce à l'ajout de nouvelles fonctionnalités et l'intégration de systèmes légataires. Il est admis que des systèmes bien conçus tendent à réduire les efforts de maintenance et donc à réduire les coûts à long terme (Banker et al., 1993; Oman et Hagemeister, 1992). Cependant, concevoir de tels systèmes est devenu bien plus compliqué depuis l'utilisation croissante d'architectures distribuées et orientées services. Afin de faciliter l'évolution et la maintenance de programmes, il est important qu'ils satisfassent au mieux de bons critères de conception et de qualité de service. Ces préoccupations furent tout d'abord évaluées dans la programmation orientée objets (OO). L'exemple le plus significatif est celui des patrons du "*Gang of Four*" (GoF) (Gamma et al., 1994), un regroupement de quatre infor-

maticiens. Ils ont proposé plusieurs bonnes pratiques, appelées patrons de conception, pour répondre à des problèmes communs et récurrents de conception logicielle. Plusieurs catalogues publiés ces dernières années fournissent aussi des patrons similaires pour le contexte des architectures orientées services (Daigneau, 2011; Erl, 2009; Rotem-Gal-Oz, 2012). L'exemple de la *Façade*, du même nom que le patron OO, correspond à un service qui cache les détails d'implémentation. L'implémentation est alors découplée du client et peut de ce fait évoluer indépendamment. Un *Routeur* est aussi un patron SOA typique (Milanovic, 2006), qui permet l'ajout d'une couche intermédiaire afin de réduire le couplage avec les services d'affaires. Bien qu'il existe des similitudes entre patrons OO et SOA, ces derniers restent différents car ils ont leurs propres propriétés structurelles et comportementales.

Plusieurs approches intéressantes ont été proposées afin d'évaluer la qualité des systèmes logiciels. Un certain nombre d'entre elles se concentrent justement sur la détection automatique de patrons de conception dans les systèmes OO (Antoniol, Fiutem et Cristoforetti, 1998; De Lucia et al., 2010; Guéhéneuc et Antoniol, 2008; Heuzeroth et al., 2003; Hu et Sartipi, 2008). Ces approches sont soit statiques, soit dynamiques, par exemple basées sur la fouille des traces d'exécution afin d'en déduire certains styles architecturaux. Elles fournissent donc un moyen sûr et cohérent d'évaluer la qualité des systèmes OO. Malheureusement, à notre connaissance, aucune approche concrète de détection de patrons n'existe dans le contexte SOA ; c'est pourquoi nous explorons ce domaine. Les travaux les plus liés correspondent à ceux réalisés par Moha et al. (Moha et al., 2012) visant la détection d'anti-patrons SOA, qui correspondent à de mauvaises pratiques de conception dans les systèmes SOA. Un langage spécifique au domaine (DSL : *Domain Specific Language*), fourni par la plateforme "*Service Oriented Framework for Analysis*" (SOFA : `http://sofa.uqam.ca`), permet la spécification de mauvaises pratiques via un vocabulaire expressif de haut niveau d'abstraction. Chaque anti-patron, dérivé de la littérature, est spécifié à l'aide d'une carte de règles, qui correspond à un ensemble de règles qui utilisent des métriques spécifiques. Ces métriques sont soit statiques, dans ce cas elles donnent des informations de type structurel comme

le couplage ou le nombre de services exposés, soit dynamiques, et donnent alors des renseignements propres à l'exécution comme le temps de réponse ou le nombre d'invocations par exemple. Un processus automatique de génération convertit les cartes de règles en algorithmes de détection, qui sont à leur tour appliqués sur les systèmes SOA à analyser.

Dans ce mémoire, nous étendons l'approche existante de la plateforme SOFA pour considérer la détection de patrons SOA durant l'exécution, pour évaluer la qualités de systèmes à base de services. Jusqu'à présent, aucune approche de détection automatique de tels patrons n'a été proposée, c'est pourquoi le travail présenté dans ce mémoire est significatif et novateur. Le travail a fait l'objet d'un article de recherche dans la conférence *ICSOC'13* (Demange, Moha et Tremblay, 2013), il est consultable en appendice A. L'approche proposée se nomme SODOP (*Service Oriented Detection of Patterns*) et conduit aux trois contributions suivantes. (1) Une analyse de domaine approfondie des différents catalogues existants nous a permis de compiler et catégoriser les meilleures pratiques dans SOA et ses implémentations sous-jacentes. (2) Cette analyse nous a conduit à la spécification de cinq patrons SOA importants à l'aide de cartes de règles. Nous les avons sélectionnés car ils représentent de bonnes pratiques, en terme de conception et de qualité de service, à la fois récurrentes et agnostiques de toute technologie. (3) La spécification des cartes de règles nous a mené à étendre la suite de métriques existante dans SOFA avec la définition de huit nouvelles métriques. Nous avons validé l'approche proposée avec deux systèmes SOA : *Home-Automation*, un système fournissant des services pour la domotique, et *FraSCAti*, une implémentation libre du standard SCA (Seinturier et al., 2009). Nous montrons que l'approche SODOP permet la spécification et la détection de patrons SOA avec de hautes valeurs de précision, de rappel et de performance.

Globalement ce mémoire est organisé de la manière suivante. Le chapitre 1 présente en détails les architectures orientées service et composant. Le chapitre 2 décrit les différentes techniques de détection de patrons de conception. Le chapitre 3 présente l'ap-

proche proposée pour la détection de patrons SOA via SODOP. Le chapitre 4 présente les détails d'implémentation de l'outil proposé. Le chapitre 5 décrit les expérimentations réalisées et les résultats obtenus sur les deux systèmes cités précédemment. Enfin, nous concluons notre travail et présentons les travaux futurs dans la dernière partie.

CHAPITRE I

SOA ET LES PATRONS DE CONCEPTION

L'apparition des systèmes informatiques et logiciels a révolutionné la façon dont le traitement des informations est effectué. Ils ont notamment permis de forts gains de productivité grâce à une automatisation toujours plus importante des processus d'affaires. Cependant, la mise en place de tels systèmes demande de nos jours une grande expertise. Elles sont toujours plus efficaces, de plus haut niveau mais aussi et surtout plus compliquées à mettre en place à la vue des retours d'expériences. Nous déterminons dans ce chapitre les objectifs actuels dans le domaine du génie logiciel afin de produire des systèmes logiciels de qualité. Nous abordons ensuite plus en détails comment les architectures de haut niveau et les processus d'assurance qualité permettent aujourd'hui d'obtenir des systèmes d'informations performants et répondant à des besoins spécifiques.

1.1 Le génie logiciel et ses objectifs

Le domaine du génie logiciel s'intéresse à définir quelles sont les bonnes pratiques nécessaires à la conception, au développement et à la maintenance de systèmes logiciels. Par définition, ce sont toutes les techniques et méthodes de construction nécessaires à l'élaboration de systèmes logiciels. Cette construction sur mesure doit avant tout répondre à des besoins spécifiques et est bornée par plusieurs critères, comme des jalons temporels ou bien les coûts prévisionnels. Le génie logiciel s'applique à toutes les phases d'un projet informatique dans le sens où il est nécessaire qu'il soit appliqué tant sur

l'analyse et la conception que sur le développement à proprement dit. Malheureusement, les statistiques font encore état de trop nombreux projets n'aboutissant pas, et ce pour plusieurs raisons tels que les dépassements de ressources budgétaires et temporelles. Le tableau 1.1 représente les différentes catégories d'aboutissement des projets informatiques au cours des deux dernières décennies.

Année d'étude	Projets réussis	Projets en difficulté	Projets échoués
1994	16%	53%	31%
1996	27%	33%	40%
1998	26%	46%	28%
2000	28%	49%	23%
2004	29%	53%	18%
2006	35%	46%	19%
2009	32%	44%	24%

Tableau 1.1 Taux de réussites des projets informatiques de 1994 à 2009.[1]

Le constat est qu'en 2009 seuls 32% des projets aboutissent dans les prévisions, 44% sont en difficulté et engendrent des dépassements de coûts et/ou délais et, surtout, que 24% d'entre eux échouent et sont stoppés avant leur terme. Cependant, une évolution encourageante se dégage : depuis 1994, le taux de projets réussis a augmenté (de 15 points) alors que les échecs de projets ont diminué. Les projets en difficulté sont, quant à eux, plutôt stables et oscillent autour de 50%.

Les causes de ces échecs ou dépassements sont nombreux et ont d'ores et déjà donné lieu à de nombreuses études. La figure 1.1 caricature bien les différents écarts possibles entre le besoin initial du client et la livraison finale du produit. Ces écarts ont de nombreuses origines : des problèmes de communication entre la maîtrise d'œuvre et

1. Études réalisées dans le rapport *CHAOS* de 2009 du *Standish Group*.

d'ouvrage, une mauvaise compréhension de l'analyste ou des erreurs de conception.

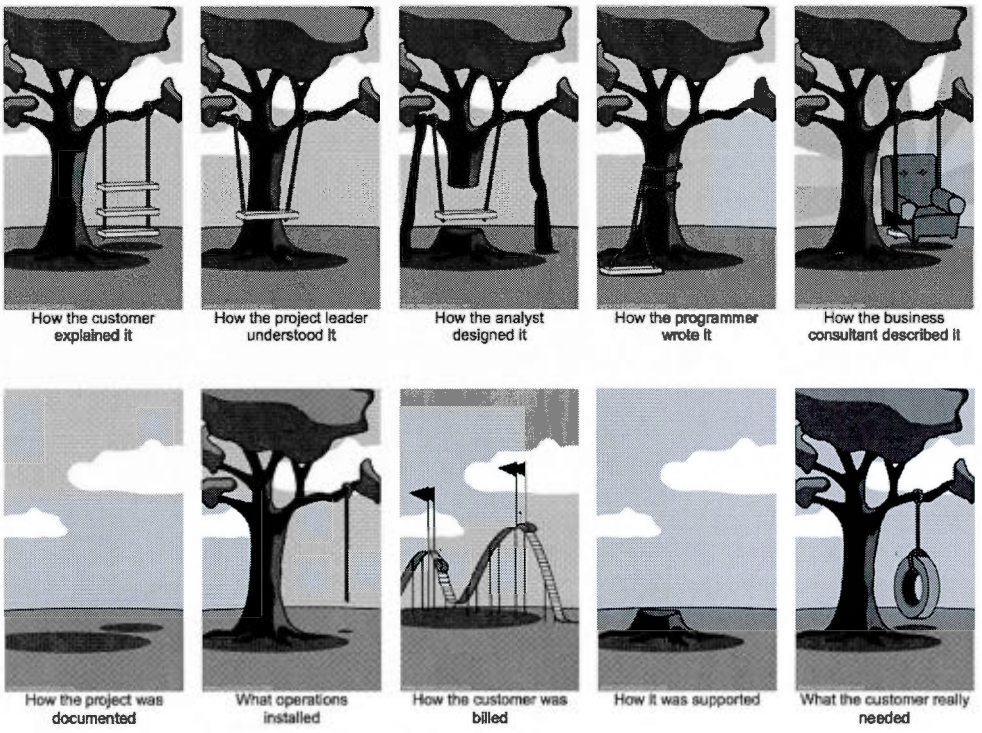

Figure 1.1 Les difficultés d'un projet informatique. [2]

Nous avons pour objectif dans ce mémoire de fournir une approche et un outil qui s'inscrivent dans une démarche spécifique d'assurance qualité qui soit expressive, fiable et performante. Nous nous intéressons particulièrement aux processus de conception ainsi qu'aux démarches d'amélioration continue de la qualité des systèmes logiciels. En effet, on remarque que la phase la plus lourde et la plus coûteuse dans le cycle de vie des logiciels est souvent la maintenance. Celle-ci, pour peu que le logiciel soit bien construit, peut être plus ou moins importante en termes de coût et de temps.

---

2. How Projects Really Work (`http://www.projectcartoon.com/`)

1.2 Les architectures orientées services et composants

Dans le but de répondre à des besoins initiaux fonctionnels de plus en plus complexes, on assiste à une évolution majeure des architectures logicielles. Les styles architecturaux ont en effet tendance à devenir de plus en plus haut niveau, c'est-à-dire de fournir un niveau d'abstraction de plus en plus élevé. Par définition, les styles architecturaux sont des guides de structuration des système logiciels permettant de répondre à des problèmes de conception spécifiques. Ils permettent notamment de décomposer physiquement et logiquement des entités logicielles, par exemple pour faciliter la réutilisation ou réduire la complexité d'une application. Au fil des décennies, nous sommes passés des systèmes monolithiques vers des systèmes en couches pour récemment voir émerger les architectures distribuées. Ces architectures distribuées donnent lieu à des systèmes logiciels ayant différents nœuds physiques distribués sur un réseau. Ces architectures migrent actuellement de simples solutions client/serveur ou n-tiers vers des architectures orientées services. En parallèle, les paradigmes de programmation ont eux aussi suivi cette tendance et sont passés de la programmation structurée (langage C) à la programmation par objets (langage Java), pour aboutir à un paradigme orienté services.

Le style architectural à base de services (SOA : *Service Oriented Architecture*) est de plus en plus adopté car il dispose de nombreux avantages. La majorité de la communauté tend à dire que le terme SOA a initialement été introduit en 1996 par *Shulte* et *Natis* dans un rapport technique paru chez *Gartner* (Schulte et Natis, 1996). Ce style architectural se base sur le concept de service, une entité logicielle autonome fournissant des interfaces standardisées pouvant être découvertes automatiquement et exécutées par des clients. Les applications construites à partir de ce style consistent à composer plusieurs services existants répondant chacun à des besoins métiers différents. Les services ont l'avantage d'être hautement réutilisables du fait de leur autonomie, leur faible couplage et leur disponibilité via un réseau. Les services ont donc le principal avantage d'être facilement réutilisés au sein de plusieurs applications. Pour finir, leur atout majeur provient du fait qu'ils soient indépendant de toutes technologies.

En effet un système à base de services peut, par exemple, être composé de langages d'implémentation tout à fait hétérogènes. Cette caractéristique permet de tirer partie des fonctionnalités existantes de systèmes légataires et de les intégrer dans des systèmes logiciels répondant à de nouvelles exigences.

L'architecture orientée composants est une des solutions permettant d'implémenter le style architectural à base de services. Elle a pour objectif de définir un ensemble de composants logiciels et de les associer pour répondre à des besoins métiers divers. Un composant est un bloc de construction implémentant une logique applicative spécifique. Il peut être implémenté via plusieurs technologies ; Java, BPEL ou C# sont les langages les plus utilisés actuellement. Ce type d'architecture peut notamment être spécifié par le standard SCA (*Service Component Architecture*) qui définit les fonctionnalités et les frontières d'une application, tout en respectant les fondements des applications orientées services. Le standard SCA a pour objectif de définir de quelle façon des composants peuvent être créés et de quelle façon ils vont se comporter à l'exécution en travaillant conjointement. Ce sont ces directives de composition qui font la grande différence entre une architecture SOA et SCA.

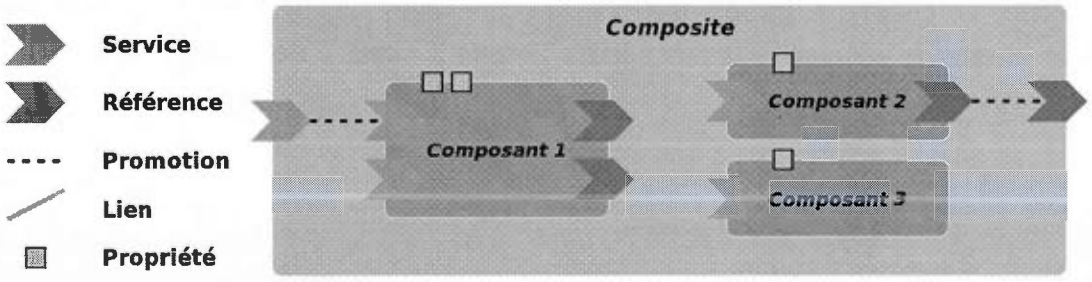

Figure 1.2 Concepts clés d'une architecture orientée composants.

La figure 1.2 présente les concepts clés d'une application orientée composants. Chaque composant expose des services, qui déclarent des méthodes potentiellement appelées par des clients, et des références vers d'autres services desquels ils dépendent.

Afin de former une application complète et fonctionnelle, il est nécessaire d'assembler un ensemble de composants existants. Cet assemblage est réalisé au moyen d'un lien de communication inter-composants (*wire*) et forme ainsi une dépendance. Le standard SCA permet aussi de réaliser des regroupements de plusieurs composants au sein d'un même composite. Ainsi ce composant peut simplement servir de couche englobante pour n'exposer que certains services ou références internes ; on dit alors qu'ils ont été promus (cf. Figure 1.2). En revanche, certains composants peuvent nécessiter des informations initiales afin d'être exécutés, et dans ce cas on parle de propriétés.

1.3 Les patrons et anti-patrons de conception

La conception de systèmes logiciels devient une tâche toujours plus complexe au vu des spécifications fonctionnelles toujours plus importantes. Développer un logiciel respectant des critères de qualité de service ainsi qu'une bonne conception est donc une tâche difficile à réaliser et nécessite une grande expertise. Plusieurs catalogues ont vu le jour afin de définir de bonnes pratiques de conception logicielle ; on parle généralement de patrons de conception. Ceux-ci furent en premier lieu énoncés pour le paradigme de programmation par objets par le *Gang of Four* (GoF, un groupe de quatre experts) dans le livre *Design Patterns* (Gamma et al., 1994). Ils y énumèrent 23 patrons de conception — bonnes pratiques de conception visant à résoudre des problèmes récurrents en programmation par objets. Ces patrons OO (orientés objets) sont catégorisés en trois groupes : création, structure et comportement, en fonction du problème auquel chacun fait référence.

Le patron *Commande* représenté sur la figure 1.3 est un exemple de patron de GoF parmi les plus utilisés. Son objectif est d'éviter un couplage fort entre deux objets, l'invocateur n'a pas connaissance du destinataire d'un événement et inversement. Dans cet exemple l'objet *Invocateur* va faire appel à la méthode *executer()* d'un objet de type *ICommande*. L'objet *Commande* va décider dynamiquement quel est le destinataire

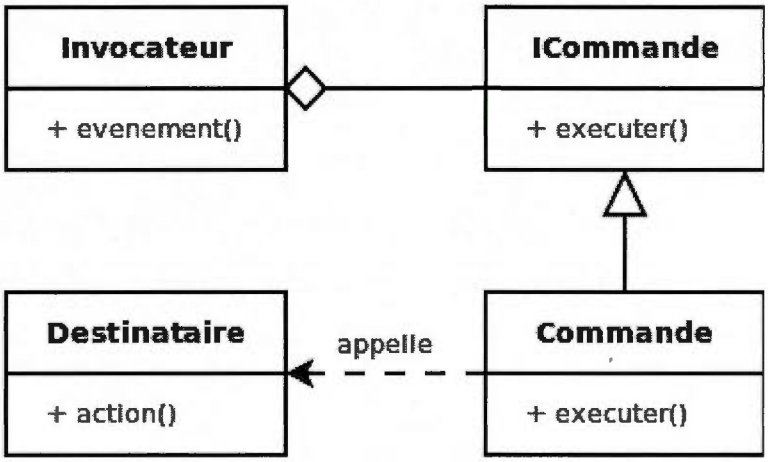

Figure 1.3 Patron *Commande* du *Gang of Four* (Gamma et al., 1994).

de l'événement pour effectuer l'action appropriée. Ce patron permet une plus grande modularité de l'application car les commandes à exécuter peuvent changer à l'exécution.

Les patrons SOA commencent eux aussi à se démocratiser malgré la jeunesse et les faibles retours d'expérience disponibles pour le moment sur les architectures SOA. Mais la question primordiale à se poser est la suivante : pourquoi ne pouvons nous pas directement appliquer les patrons définis pour le paradigme objet sur les architectures SOA ? Hohpe, dans son rapport intitulé *SOA Patterns : New Insights or Recycled Knowledge ?* (Hohpe et Easy, 2007), explique justement que SOA n'est pas simplement une nouvelle technologie à la mode. D'après lui, SOA est clairement un nouveau modèle de programmation qui nécessite des approches spécifiques notamment grâce à la spécification de nouveaux patrons. En effet, les applications orientées objets ne peuvent pas directement être comparées aux applications à base de services, car elles disposent chacune de leurs propres propriétés structurelles et de leurs propres comportements.

Cependant les catalogues définissant les patrons SOA ne sont apparus qu'à partir de 2009 (Daigneau, 2011; Erl, 2009; Rotem-Gal-Oz, 2012). Les spécifications de patrons SOA disponibles dans la littérature ne sont, pour le moment, pas toujours cohérentes. En effet, elles utilisent souvent des classifications différentes selon la nature, l'étendue

ou les objectifs des patrons définis. Après une revue de cette littérature, nous avons identifié trois groupes principaux permettant de classifier la plupart des patrons SOA disponibles :

1. *Les patrons structuraux* : ils ont pour objectif de satisfaire des préoccupations communes comme l'autonomie ou la réutilisation des services. Ce type de patron s'applique la plupart du temps directement sur un service afin de garantir une structure optimale.
2. *Les patrons d'intégration et de communication* : ils décrivent comment les services doivent être composés et orchestrés afin de répondre à des besoins métiers de haut niveau. Cette catégorie prend également en compte les types de communication disponibles entre services, comme par exemple, les échanges synchrones ou asynchrones.
3. *Les patrons de qualité de service* : ils ont pour but d'assurer des caractéristiques de qualité de service spécifiques comme la mise à l'échelle, la performance ou les exigences de sécurité.

Thomas Erl est l'auteur d'un des catalogues les plus complets avec la spécification de plus de 80 patrons SOA (Erl, 2009). Chaque patron est détaillé suivant un protocole précis : il commence par décrire le problème à résoudre, puis le patron SOA permettant d'y répondre. Le plus intéressant est le fait qu'il énumère les situations où l'appliquer ainsi que les situations dans lesquelles le patron peut impacter l'application.

Le patron *Service Decomposition* présenté à la figure 1.4 est un exemple typique de décomposition de service dans une architecture SOA (Erl, 2009). Ce patron s'applique lorsqu'un service est de granularité trop forte, c'est à dire qu'il répond à des besoins métiers trop nombreux, ce qui nuit à sa réutilisation. La solution indiquée est de décomposer ce service en plusieurs sous-services restreints à des fonctionnalités métiers plus spécifiques et plus cohésives. Dans le cas présent, le service *Invoice* serait décomposé

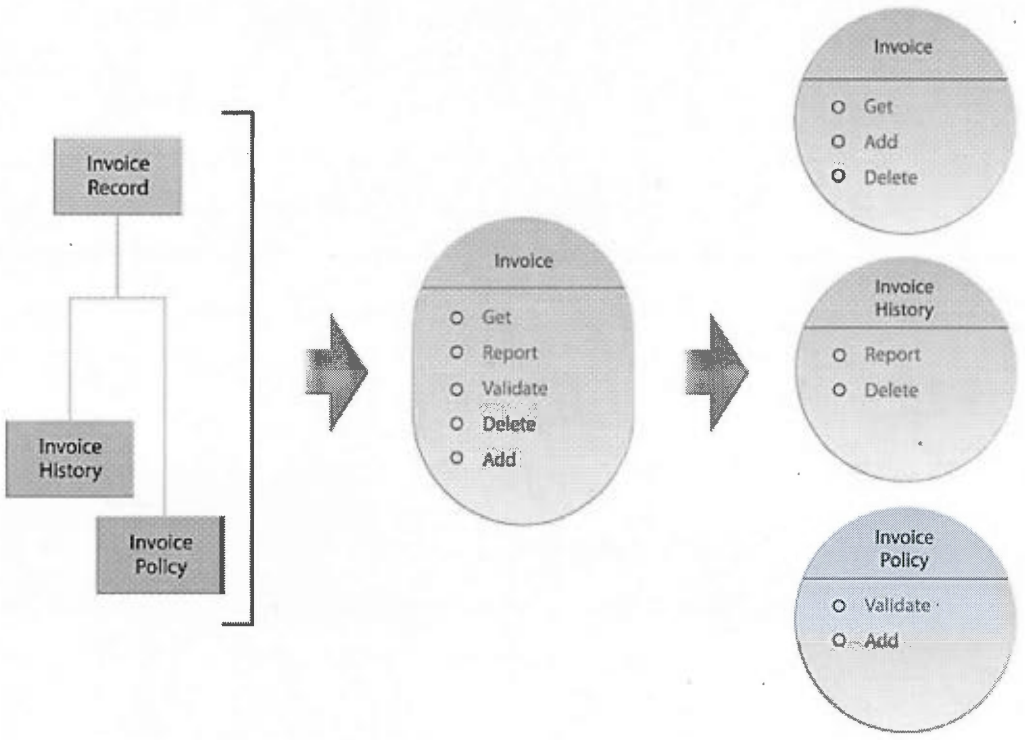

Figure 1.4 Patron SOA *Service Decomposition* (Erl, 2009).

en trois sous-services. *Invoice* servirait uniquement à récupérer, ajouter ou supprimer une commande, alors que *Invoice History* et *Invoice Policy* seraient respectivement responsables de l'archivage et de la validation des commandes. Cette solution permet alors une meilleure réutilisation de chaque service au sein d'autres applications, cependant cela implique des compositions plus complexes à mettre en œuvre.

Les anti-patrons, à l'opposé des patrons, font référence à de mauvaises pratiques de conception qu'il faut absolument éviter dans les systèmes logiciels. Le terme a vraisemblablement été introduit par *Andrew Koenig* dans l'article *Patterns and Antipatterns* (Koenig, 1995) pour le paradigme de programmation à objets. La littérature spécifiant les anti-patrons OO est assez importante, et plusieurs mauvaises pratiques ont déjà été identifiées ces dernières années. L'anti-patron *référence cyclique* (*Circular Dependency*) est un exemple typique : il suggère qu'il faut éviter que deux objets soient

dépendants l'un de l'autre. L'anti-patron *objet omniscient* (*God Object*) est lui aussi très connu : il ne faut pas assigner trop de responsabilités au même objet, pour faciliter son évolution et sa réutilisation.

Les anti-patrons SOA ne sont pour le moment que rarement spécifiés dans la littérature (Dudney et al., 2003; Moha et al., 2012; Rotem-Gal-Oz, 2012). Cependant, ils partagent souvent des caractéristiques communes avec certains anti-patrons du paradigme objet. Le *Nano Service* (Rotem-Gal-Oz, 2012), aussi connu sous le nom de *Tiny Service* (Dudney et al., 2003), est représenté à la figure 1.5.

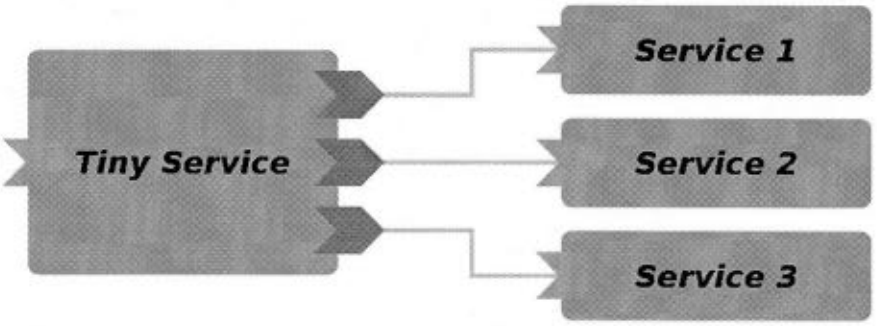

Figure 1.5 Anti-patron SOA *Tiny Service* (Dudney et al., 2003).

Cet anti-patron est à l'opposé du *Multi Service* (Dudney et al., 2003), il se caractérise par un faible nombre de méthodes exposées. Un service conçu de cette façon ne répond pas à une logique applicative complète et ne réalise qu'une partie d'une abstraction. Il induit donc une plus grande complexité dans la composition des services afin de fournir une application répondant aux besoins fonctionnels. Nous remarquons que la conception des services d'une architecture SOA est souvent complexe. Elle doit tenir compte d'un certain nombre de caractéristiques qui permettent de faciliter la maintenance et l'évolution de systèmes SOA.

CHAPITRE II

TECHNIQUES DE DÉTECTION DE PATRONS

Plusieurs techniques permettent actuellement d'évaluer la qualité des systèmes logiciels. La satisfaction des besoins fonctionnels du client n'est pas l'unique caractéristique de qualité à considérer. Celle-ci ne prend pas en compte les critères de qualité de service et de qualité de conception, permettant à terme, par exemple de réduire les coûts de maintenance. La détection de patrons de conception est une des techniques utilisées en génie logiciel pour évaluer la bonne conception d'une application selon plusieurs métriques de qualité. Dans les sections suivantes, nous présentons les techniques de détection utilisées tant dans les programmes objets que les applications SOA.

2.1 Analyse du code source

La détection automatique de patrons a déjà été expérimentée dans le paradigme de programmation orienté objets qui est beaucoup plus mature et très utilisé actuellement. *Antoniol et al.* ont fait partie des premiers à proposer une approche de découverte de patrons dans les programmes objets (Antoniol, Fiutem et Cristoforetti, 1998). La première étape de leur approche consiste à associer le code source d'une application vers un langage abstrait intermédiaire permettant d'être indépendant du langage à objets utilisé. Durant la seconde étape, de nombreuses métriques statiques, comme le nombre d'attributs, de méthodes, d'associations, sont calculées sur ce langage intermédiaire. Le processus final vise à découvrir les patrons introduits dans la phase de conception durant une détection automatique. Les similitudes entre chaque représentation intermédiaire

permet d'identifier les patrons de *GoF* présents dans l'application. La figure 2.1 est la représentation du patron OO *Adaptateur* dans ce langage. On y distingue notamment la définition des trois classes *Target*, *Adapter* et *Adaptee* ainsi que leurs méthodes. La relation d'héritage entre *Target* et *Adapter* est spécifiée dans le bloc *GENERALIZATION*. Le dernier bloc *RELATION* permet de définir les associations et cardinalités entre les classes spécifiées.

```
CLASS  Target
        OPERATIONS
                PUBLIC Request();
CLASS  Adapter
        OPERATIONS
                PUBLIC Request();
CLASS  Adaptee
        OPERATIONS
                PUBLIC SpecificRequest();
GENERALIZATION Target
        SUBCLASSES Adapter;
RELATION Delegates
        ROLES
                CLASS Adapter MULT One,
                CLASS Adaptee MULT One
```

Figure 2.1 Représentation du patron *Adaptateur* via un langage intermédiaire.

Rasool et al. ont également proposé une approche intéressante de détection de patrons basée sur des annotations, des requêtes SQL et des expressions régulières (Rasool, Philippow et Mäder, 2010). Les annotations ont en effet l'avantage de pouvoir représenter les intentions du développeur et ainsi fournir un support d'information pour la détection de patrons. Leur première étape consiste à annoter manuellement le code source avec certains concepts de programmation OO (abstraction, héritage, composition, etc.). La rétro-ingénierie du code source permet ensuite de générer un modèle de code source incluant les relations entre les différents composants. En parallèle, chacun des patrons de GoF est transcrit en règles puis en requêtes SQL et expressions régulières. Un processus final permet d'appliquer les requêtes SQL et expressions régulières sur le

modèle de code source afin d'en extraire les candidats à chaque patron.

Ces deux approches ont montré que la découverte de patrons de conception basée sur l'analyse et l'instrumentalisation de code source est efficace. Cependant elles ont, tout comme beaucoup d'autres travaux similaires, l'inconvénient d'être seulement statiques. Il n'est, par exemple, pas possible de réaliser une détection de patrons comportementaux avec ces approches. De plus la nature des systèmes à base de services ne permet pas, dans la majorité des cas, d'instrumenter le code source de leur implémentation dans le but d'y détecter des patrons. C'est pourquoi de nombreuses approches se concentrent sur la manipulation des traces d'exécution, qui peuvent être mises à disposition dans certains cas.

2.2 Fouille des traces d'exécution

Tsantalis et al. ont justement pallié la lacune mentionnée dans la section précédente, en proposant une approche novatrice de détection de patrons comportementaux (Tsantalis et al., 2006). Leur approche de détection est réalisée via un processus de fouille d'associations, basé sur les traces d'exécution d'un système. Le processus consiste à extraire des graphes, puis des matrices, pour chacun des concepts objets suivants : association, héritage, classe abstraite et appel de méthode abstraite. En se basant sur la définition de patrons de *GoF* de la littérature, chaque patron est transcrit en différentes matrices. Un algorithme basé sur un score de similarité permet ensuite d'identifier les candidats probables à chaque patron. Cette approche a l'avantage de considérer des comportements plus dynamiques et ainsi détecter des implémentations qui varient d'un patron de base.

Ka-Yee Ng et al. ont proposé une solution alternative, basée sur un processus de détection dynamique de patrons de conception (Ka-Yee Ng, Guéhéneuc et Antoniol, 2007). L'étape initiale consiste à spécifier un diagramme de scénarios pour chaque patron à considérer dans la détection. Ils utilisent ensuite un processus de rétro-ingénierie basé sur les traces d'exécution afin de construire un diagramme de scénarios pour l'application

en cours d'analyse. Le diagramme de scénarios de l'application est ensuite comparé au diagramme de scénarios de chaque patron, via un processus de programmation par contraintes, pour y identifier les candidats potentiels.

Wendehals et al. ont combiné le meilleur de chacun des travaux précédents pour fournir une approche à la fois statique et dynamique afin de détecter tant des patrons structuraux que comportementaux (Wendehals et Orso, 2006). Leur approche consiste à transformer l'ensemble des appels de l'application en un automate à états finis. Un processus de correspondance entre ces automates et les modèles de chaque patron donnent les candidats les plus probables.

À notre connaissance, il n'existe que peu de travaux permettant d'évaluer la qualité de service et la qualité de conception d'applications à base de services. *Yousefi et al.* ont récemment proposé une approche permettant la détection de fonctionnalités spécifiques en instrumentant les traces d'exécution de systèmes SOA (Yousefi et Sartipi, 2011). Un gestionnaire leur permet dans un premier temps d'exécuter des scénarios d'exécution prédéfinis, afin d'en récupérer les traces distribuées sur chaque hôte. Un processus additionnel de filtrage, permettant l'élimination du bruit et des appels omniprésents, permet d'obtenir des traces plus expressives. Ils utilisent ensuite un algorithme de fouille de données, de bas en haut, appliqué sur les traces, afin de construire des graphes d'ensemble d'éléments fréquents fermés. Ce processus permet l'extraction de fonctionnalités spécifiques à certains scénarios en se basant sur la fréquence d'appels. Les résultats obtenus ont pour objectif de faciliter la maintenance de l'application en donnant priorité aux services fournissant les fonctionnalités les plus importantes.

2.3 Détection à l'exécution

La détection de patrons de conception à l'exécution est une troisième et dernière possibilité permettant de réaliser une détection de patrons sur un système logiciel. Cette technique a l'avantage d'être réalisée directement sur un système, ce qui permet donc de s'affranchir des tâches de collecte et de filtrage des traces d'exécution. *Moha et al.*

ont proposé l'approche SODA (*Service Oriented Detection of Antippatterns*) pour la spécification et la détection d'anti-patrons SOA (Moha et al., 2012). Ils sont, à notre connaissance, les seuls à fournir ce type d'approche dans un environnement à base de services. C'est sur cette approche que nos travaux de recherches ont été menés, et elle est décrite dans le chapitre 3.

CHAPITRE III

SPÉCIFICATION ET DÉTECTION DE PATRONS SOA AVEC SODOP

Nous proposons l'approche SODOP (*Service Oriented Detection Of Patterns*) qui vise la spécification manuelle et la détection automatique de patrons SOA. SODOP est une extension de l'approche précédente proposée par *Moha et al.* appelée SODA (*Service Oriented Detection of Antipatterns*) (Moha et al., 2012). Dans les sections suivantes, nous présentons SODA pour comprendre les processus de détection utilisés ainsi que les évolutions réalisées. Enfin, nous présentons l'approche SODOP ainsi que les cinq patrons SOA que nous avons spécifiés.

3.1 Rappels de l'approche SODA

SODA est une approche en trois étapes pour la détection d'anti-patrons. La première étape consiste en la spécification d'anti-patrons SOA en utilisant un langage spécifique au domaine (DSL : *Domain Specific Language*). Ce DSL permet de définir des cartes de règles, qui sont des ensembles de règles permettant de décrire des propriétés de conception ou de qualité de service spécifiques. La figure 3.1 montre la grammaire du DSL, sous la forme Backus-Naur. Une carte de règles spécifie une règle ou une combinaison de plusieurs règles (ligne 3) en utilisant des opérateurs ensemblistes (ligne 6). Une règle fait référence soit à une métrique, soit à une carte de règles déjà existante (ligne 5). Une métrique peut être soit statique (ligne 10), soit dynamique (ligne 11) — calculée à l'exécution. Les exemples de métriques statiques incluent le nombre de méthodes déclarées (NMD : *Number of Methods Declared*) ou le nombre de références sortantes

```
1  rule_card      ::= RULE_CARD: rule_card_name { (rule)+ };
2  rule           ::= RULE: rule_name { content_rule };
3  content_rule   ::= metric | set_operator rule_type (rule_type)+
4                   | RULE_CARD: rule_card_name
5  rule_type      ::= rule_name | rule_card_name
6  set_operator   ::= INTER | UNION | DIFF | INCL | NEG
7  metric         ::= metric_value comparator (metric_value | ordi_value
                    | num_value)
8  metric_value   ::= id_metric (num_operator id_metric)?
9  num_operator   ::= + | - | * | /
10 id_metric      ::= ANAM | ANIM | ANP | ANPT | COH | NID
                    | NIR | NMD | NOR | NSC | TNP
11                  | A | DR | ET | NDC | NIC | NOC
                    | NTMI | POPC | PSC | SR | RT
12 ordi_value     ::= VERY_LOW | LOW | MEDIUM | HIGH | VERY_HIGH
13 comparator     ::= < | ≤ | = | ≥ | >
14  rule_cardName, ruleName ∈ string
15  num_value ∈ double
```

Figure 3.1 Grammaire BNF des cartes de règles.

(NOR : *Number of Outgoing References*). Les exemples de métriques dynamiques incluent le temps de réponse (RT : *Response Time*) ou le nombre d'appels entrants (NIC : *Number of Incoming Calls*). Chaque métrique peut être comparée à des valeurs ordinales — une échelle de Lickert de cinq valeurs allant de très bas (*very low*) à très haut (*very high*) (ligne 12). Elle peut aussi être comparée à une valeur numérique (ligne 8) à l'aide de comparateurs arithmétiques (ligne 13). Une valeur de métrique est calculée pour chaque service par rapport à tous les autres pour peupler une "boîte à moustache" (*box-plot*). La figure 3.2 décrit comment les valeurs ordinales sont associées aux

intervalles de la boîte à moustache. Les valeurs aberrantes très basses et très hautes correspondent respectivement aux intervalles très bas et très haut. Les valeurs comprises d'une part entre le minimum et le premier quartile et d'autre part entre le troisième quartile et le maximum forment respectivement les intervalles bas (*low*) et haut (*high*). Enfin les valeurs restantes comprises entre le premier et le troisième quartile forment le dernier intervalle intermédiaire (*medium*).

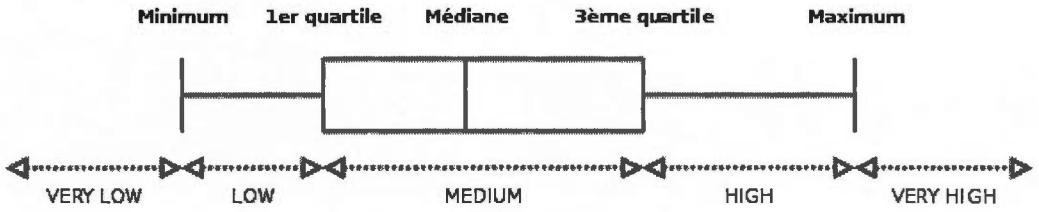

Figure 3.2 Association entre valeurs ordinales et intervalles de la boîte à moustache.

La seconde étape consiste à générer automatiquement les algorithmes de détection correspondant aux cartes de règles initialement spécifiées. Ceux-ci sont générés en se basant sur un méta-modèle EMF (Eclipse Modeling Framework Project, 2011) combiné avec l'outil de génération de code Acceleo (Acceleo Code Generator Tool, 2010). La troisième et dernière étape consiste à appliquer ces algorithmes sur des systèmes SOA dans le but d'y détecter les services candidats aux anti-patrons spécifiés.

3.2 L'approche SODOP

La flexibilité de SODA a permis d'étendre l'approche de détection d'anti-patrons à celle de patrons SOA. En effet, le DSL et la plateforme SOFA (*Service Oriented Framework for Analysis*) associée permettent aisément la définition de nouvelles règles et l'intégration de nouvelles métriques. L'approche proposée ici, du nom de SODOP (*Service Oriented Detection Of Patterns*), introduit la spécification de cinq patrons SOA que nous avons tirés de la littérature (Erl, 2009; Rotem-Gal-Oz, 2012). Ces patrons

ont été spécifiés à l'aide des cartes de règles, comme décrit dans l'approche SODA, en combinant les métriques existantes auxquelles huit nouvelles métriques ont été ajoutées ; celles-ci sont soulignées sur la figure 3.1. L'approche en trois étapes utilisée dans SODOP, présentée à la figure 3.3, reste semblable à celle de SODA mais est spécifique aux patrons SOA.

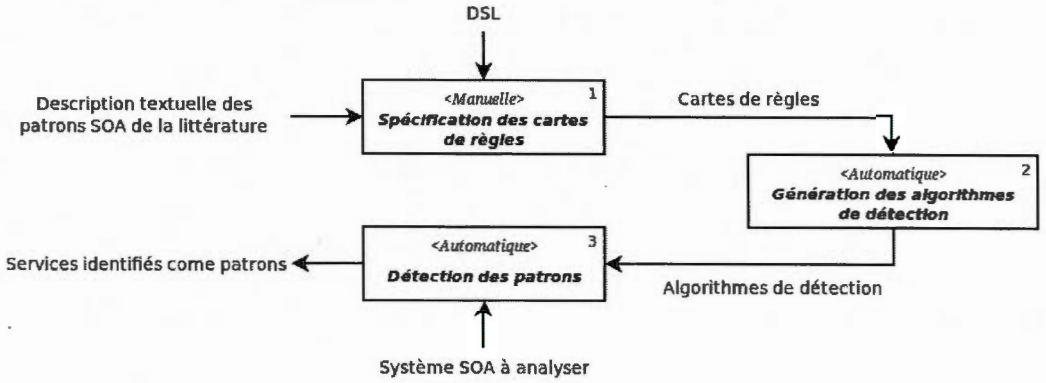

Figure 3.3 Les trois étapes de l'approche SODOP.

Les descriptions textuelles des patrons SOA ainsi que le DSL permettent la spécification manuelle des cartes de règles lors de la première étape. Lors de la deuxième étape, une génération automatique de code permet la transformation de chaque carte de règles en algorithmes de détection. Ces algorithmes sont enfin appliqués sur les systèmes SOA à analyser dans le but d'y détecter des services en tant que patrons. Nous présentons les évolutions introduites sur chacune des étapes dans les sous-sections suivantes.

3.2.1 Évolution de la grammaire des cartes de règles

La grammaire du DSL, présentée à la figure 3.1 (page 22), a été grandement simplifiée et étendue pour permettre une plus grande flexibilité dans la spécification des cartes de règles. Nous avons ajouté la possibilité de combiner plusieurs métriques existantes via des opérateurs numériques (lignes 8 et 9) pour permettre de définir des métriques plus complexes. Ainsi, nous pouvons par exemple proposer une règle comme

un ratio de deux métriques déjà existantes. L'expression 3.1 présente les différentes possibilités de combinaisons de deux métriques M_1 *et* M_2 pour une règle X.

$$X = M_1 \mid M_1 + M_2 \mid M_2 - M_1 \mid M_1 * M_2 \mid \frac{M_1}{M_2} \tag{3.1}$$

Auparavant, une métrique pouvait uniquement être comparée à des valeurs numériques ou ordinales à l'aide d'opérateurs de comparaison classiques tels que supérieur (>), égal (=), etc. Il est désormais aussi possible de comparer deux métriques (simples ou complexes) entre elles, comme leur égalité par exemple. Les trois comparaisons applicables aux métriques sont décrites par l'expression 3.2.

$$RULE = X > 1.0 \mid X \leq LOW \mid X = Y \tag{3.2}$$

Nous avons aussi décidé de retirer les relations d'inclusion dans l'ancien DSL qui n'étaient pas assez génériques selon nous. Nous voulons garder une grammaire applicable aux architectures SOA, donc sans notions de compositions explicites propres aux architectures SCA.

3.2.2 Spécifications des nouvelles métriques

Les nouvelles métriques introduites dans la plateforme SOFA sont soulignées sur la figure 3.1 (ligne 11) et sont détaillées ci-dessous.

ET : Execution Time

Le temps d'exécution (ET) représente le temps requis par un service pour réaliser sa tâche. Le temps d'exécution diffère du temps de réponse (RT) car il exclut le temps d'exécution des services imbriqués.

NDC : Number of Different Clients

Le nombre de clients différents (NDC) comptabilise le nombre distinct de clients

qui accèdent à un composant. Les appels réalisés plusieurs fois par le même client ne sont donc pas comptabilisés via cette métrique.

NIC/NOC : Number of Incoming/Outgoing Calls

À l'inverse, le nombre d'appels entrants (*NIC*) et le nombre d'appels sortants (*NOC*) représentent des appels dynamiques, donc comptabilisant potentiellement plusieurs fois le même client.

DR : Delegation Ratio

Le ratio de délégation (*DR*) représente le ratio entre appels entrants et sortants que le composant relaie à un autre service.

SR : Service Reuse

Le taux de réutilisation du service (*SR*) est une métrique dynamique qui calcule jusqu'à quelle mesure un service est réutilisé. Plus précisément, cette métrique correspond en réalité au ratio entre le nombre d'appels entrants (NIC) et le nombre total d'appels dans le système.

POPC : Proportion of Outgoing Path Change

La proportion de chemins de sortie différents (*POPC*) représente la proportion d'appels sortants dont le chemin change pour un même appel de méthode entrant. En d'autres termes, cette proportion est nulle si le chemin de sortie reste toujours le même pour un appel de méthode d'un service donné.

PSC : Proportion of Signature Change

La proportion de changement de signatures (*PSC*) calcule la proportion de changement de signatures de méthodes entre appels entrants et sortants. Cette métrique représente le niveau de différence textuelle entre la signature de la méthode entrante et celle de la méthode sortante. Nous avons calculé cette différence via la distance de similarité de Jaro-Wrinkler (Winkler, 1990) entre deux noms de méthodes. Nous avons choisi de raisonner sur les informations textuelles des méthodes car nous ne pouvons pas directement instrumenter l'implémentation d'une méthode.

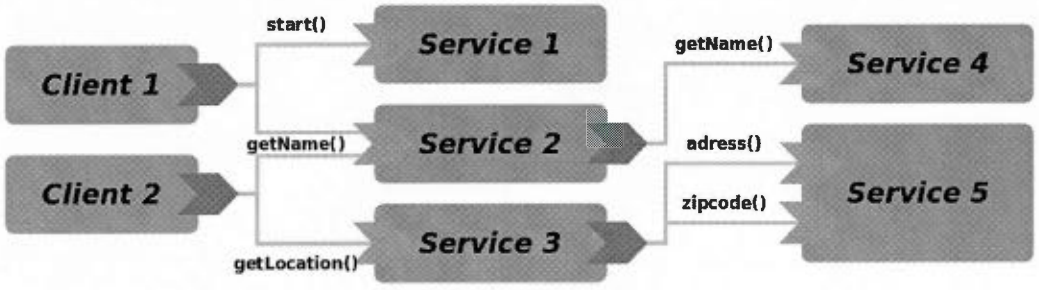

Figure 3.4 Exemple d'application SOA.

Exemple

Afin de bien comprendre l'utilité de chaque métrique introduite, nous prenons l'exemple de l'application SOA de la figure 3.4. Celle-ci dispose de cinq services, exposant un total de six méthodes décrivant une fonctionnalité précise. Nous prenons ensuite un scénario avec deux clients différents permettant d'impliquer chacun des services. La figure 3.5 représente la pile d'appels de chacune des méthodes de services, de manière à reconstituer un journal d'exécution. Le premier nombre représente l'empreinte d'horodotage de l'appel (date précise), suivi de l'émetteur et destinataire (qui varie en fonction du fléchage), et enfin la méthode impliquée. On constate que le *Service 2* est dépendant du *Service 4*, de même que le *Service 3* est dépendant du *Service 5*.

À partir de la pile d'exécution, nous avons calculé la valeur de chacune des métriques dynamiques présentées et nous les avons reportées dans le tableau 3.1. Les services 2, 3 et 5 sont les seuls à être appelés deux fois (NIC = 2), le *Service 2* est l'unique à posséder deux clients différents. Nous pouvons aussi constater que les services 2 et 3 sont dépendants, ils exposent des références et nécessitent certains appels imbriqués pour répondre au besoin fonctionnel. Le *Service 2* ne relaie les appels qu'une fois sur deux (DR = $\frac{1}{2}$), alors que le *Service 3* délègue dans tous les cas (DR = 1). La réutilisation de chaque service est calculée directement par rapport à la proportion du nombre d'appels total, les services 2, 3 et 5 sont donc les plus utilisés. On constate que le relais du *Service 2* est, dans notre cas, toujours à destination du *Service 4*

1382990853 - Client1 → Service1.start();
1382990870 - Client1 ← Service1.start();
1382990900 - Client1 → Service2.getName();
1382991250 - Service2 → Service4.getName();
1382991300 - Service2 ← Service4.getName();
1382991400 - Client1 ← Service2.getName();
1382991550 - Client2 → Service2.getName();
1382991580 - Client2 ← Service2.getName();
1382991600 - Client2 → Service3.getLocation();
1382991655 - Service3 → Service5.address();
1382991863 - Service3 ← Service5.address();
1382991960 - Client2 ← Service3.getLocation();
1382992130 - Client2 → Service3.getLocation();
1382992250 - Service3 → Service5.zipcode();
1382993781 - Service3 ← Service5.zipcode();
1382993901 - Client2 ← Service3.getLocation();

Figure 3.5 Pile d'appels du scénario d'exécution d'exemple.

(POPC = 0), et plus spécifiquement sur la même signature de méthode (PSC = 0). Le *Service 3* fait exception puisque la délégation de ses appels entrants est réalisée vers des services différents (POPC = 1) ainsi que sur des signatures totalement différentes (PSC = 1). Ces deux dernières métriques ne sont pas applicables aux autres services car ils ne délèguent jamais les appels dans ce scénario. Pour finir, on constate que le service indéniablement le plus lent en termes d'exécution est le *Service 5* (870 ms) ; à titre de comparaison le service 1 est 50 fois plus rapide (17 ms).

ServiceName	ET	NDC	NIC	NOC	DR	SR	POPC	PSC
Service 1	17 ms	1	1	0	0	$\frac{1}{8}$	-	-
Service 2	240 ms	2	2	1	$\frac{1}{2}$	$\frac{2}{8}$	0	0
Service 3	196 ms	1	2	2	1	$\frac{2}{8}$	1	1
Service 4	50 ms	1	1	0	0	$\frac{1}{8}$	-	-
Service 5	870 ms	1	2	0	0	$\frac{2}{8}$	-	-

Tableau 3.1 Valeurs des nouvelles métriques pour le scénario d'exemple.

3.3 Spécification de patrons SOA

Dans les sections suivantes, nous commençons par décrire les cinq patrons SOA que nous avons identifiés dans la revue de la littérature présentée à la section 1.3. Nous décrivons premièrement les objectifs principaux de chacun des patrons, en plus de spécifier leur architecture générique à l'aide d'un schéma. Nous donnons ensuite la carte de règles spécifiée pour chacun d'eux en explicitant l'utilisation des métriques employées.

3.3.1 Patron Optimal

Lorsque nous nous sommes intéressés à la spécification et la détection de tels patrons, nous nous sommes rendus compte qu'un certain nombre de bonnes pratiques logicielles devaient être suivies. Nous voulions pouvoir spécifier les meilleures caractéristiques fondamentales que chaque architecte ou concepteur devrait suivre lors de son analyse. Nous sommes arrivés à la conclusion qu'un patron répondant à ces critères pouvait représenter les pratiques optimales en termes de conception de services. Plusieurs principes, dont certains extraits du livre *SOA Patterns* (Erl, 2009), ont été pris en compte pour la spécification. La réutilisabilité des composants de même qu'une forte cohésion, font parties des exigences communes dans la conception de systèmes informatiques, notamment dans les programmes OO (Basili, Briand et Melo, 1996). La nature dynamique

des systèmes SOA introduit de nouvelles exigences comme la haute disponibilité (A) ou le faible temps de réponse (RT).

Ces métriques sont combinées dans une carte de règles décrite sur la figure 3.10(a) (page 34) pour la spécification de ce patron optimal. La ligne 1 permet la définition de la nouvelle carte de règles de ce patron optimal (*Basic Service*). La ligne 2 permet de spécifier que cette carte utilise l'intersection (*INTER*) de quatre règles :

- *HighSR* : Service ayant une réutilisation (*SR*) forte ou très forte, donc supérieure au troisième quartile de la boîte à moustache.
- *HighCOH* : Service dont la cohésion (*COH*) est forte ou très forte.
- *HighA* : Service dont la disponibilité (*A*) est forte ou très forte.
- *LowRT* : Service dont le temps de réponse (*RT*) est faible ou très faible.

3.3.2 Patron Façade

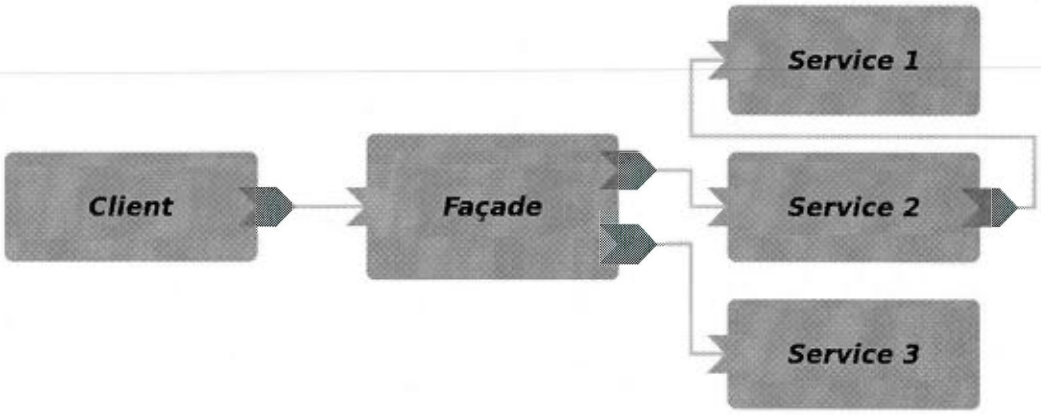

Figure 3.6 Patron Façade.

Une façade, comme illustrée sur la figure 3.6, est utilisée dans les systèmes SOA pour obtenir un plus haut niveau d'abstraction entre les couches consommateur et fournisseur. *Fowler* et *Erl* décrivent des variantes de ce patron qu'ils nomment respectivement *Remote Facade* (Fowler, 2002) et *Decoupled Contract* ou *Service Decomposition* (Erl, 2009). Ils donnent, entre autres, ce patron comme exemple dans un processus

d'intégration de systèmes légataires dans des systèmes orientés services. Ce patron est similaire au patron Façade de *GoF* dans le sens où il permet de cacher des détails d'implémentation (Gamma et al., 1994), tels que des appels imbriqués. Il permet aussi de réduire le couplage entre le client et les services sous-jacents, de telle sorte que leur implémentation puisse évoluer indépendamment, sans briser le contrat avec le client. Autrement dit, ce patron permet de masquer une décomposition complexe d'un système SOA et permet de conserver le principe de la *séparation des préoccupations*. Il sera donc plus aisé de réutiliser chaque composant ou couche dans d'autres systèmes logiciels.

Les responsabilités d'une façade peuvent être variées, mais en général elle se retrouve souvent dans les mécanismes d'orchestration dans le but de décrire comment la composition de multiples services permet de répondre à des besoins métiers spécifiques. Étant donné qu'une façade joue le rôle de couche de présentation et de communication envers différents clients, nous la caractérisons avec un haut temps de réponse (RT). Puisque ce patron cache des détails d'implémentation, son ratio d'appels entrants/sortants (NIC/NOC) est faible, donc un tel service tend à avoir plusieurs appels sortants pour un entrant. Pour finir, nous supposons qu'un tel service a un haut ratio de délégation (DR) car il ne fournit pas de logique métier spécifique mais au contraire transfère chaque appel. La figure 3.10(b) montre la carte de règles spécifiée pour le patron Façade.

3.3.3 Patron Proxy

Figure 3.7 Patron Proxy.

Le patron Proxy, représenté sur la figure 3.7, est aussi un patron de conception objet très connu, qui ajoute une indirection entre un client et un service destinataire. Son objectif diffère de la façade, car il peut par exemple ajouter de nouveaux com-

portements, la plupart du temps non-fonctionnels. Ceux-ci peuvent être utilisés à des fins de sécurité comme la confidentialité, l'intégrité voire même la journalisation des appels pour garantir une traçabilité. Il existe différents types de Proxy : *Service Interceptor* (Daigneau, 2011) et *Service Perimeter Guard* (Erl, 2009) en sont des exemples. De ce fait, il existe plusieurs possibilités quant à leur spécification sous forme de carte de règles. Nous avons choisi de favoriser une version plus générique avec les caractéristiques suivantes. La proportion entre appels entrants et sortants (NIC/NOC) doit être égale à 1 car ce patron ne sert que de relais. De plus, les propriétés de ce patron impliquent que les relais se fassent en général sur les mêmes signatures de méthodes ; sa proportion de changement de signatures (PSC) est donc basse. Un tel patron ajoute la plupart du temps des fonctionnalités communes à plusieurs services et est donc l'un des plus utilisés de l'application, son taux de réutilisation est donc haut comparé aux autres services (SR). La figure 3.10(c) montre la carte de règles spécifiée pour le patron Proxy.

3.3.4 Patron Adaptateur

Figure 3.8 Patron Adaptateur.

Le patron Adaptateur, sur le figure 3.8, partage lui aussi des caractéristiques communes avec le patron de *GoF* du même nom. Son but est d'adapter un appel de service entrant à un service de destination. L'intégration de systèmes légataires dans des architectures SOA est un exemple récurrent de son utilisation. Il permet l'adaptation des différentes interfaces d'appels (API) afin d'éviter des modifications majeures des systèmes d'ores et déjà fonctionnels. Daigneau donne l'exemple du patron *Datasource Adapter* (Daigneau, 2011) comme solution d'accès aux données pour des plateformes hétérogènes. En général, son ratio d'appels entrants/sortants (NIC/NOC) est aussi égal à 1, tout comme un Proxy. Cependant son rôle fondamental est l'adaptation, sa pro-

portion de changement de signatures (PSC) entre appels est donc élevée. C'est cette caractéristique qui va le différencier du patron Proxy qui, lui, préserve les signatures. La figure 3.10(d) montre la carte de règles spécifiée pour le patron Adaptateur.

3.3.5 Patron Routeur

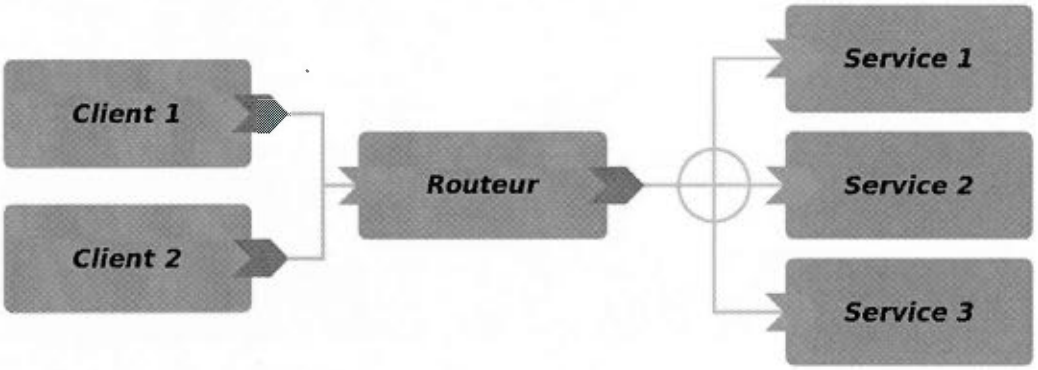

Figure 3.9 Patron Routeur.

La patron Routeur, comme illustré sur la figure 3.9, s'apparente à un routeur réseau si l'on s'en tient à ses responsabilités d'aiguillage de paquets sur un réseau. Un routeur SOA distribue les appels entrants vers différentes destinations en fonction de différents critères, comme l'identité du client ou bien les paramètres de l'appel. Certains routeurs SOA intelligents vont même jusqu'à détecter le meilleur chemin possible en fonction de critères dynamiques comme la disponibilité ou bien l'historique des appels. Afin de détecter ce type de patron, nous nous sommes concentrés sur la métrique dynamique particulière qu'est la proportion de changement de chemins de sortie. Plus un appel entrant va disposer de sorties différentes (POPC), plus nous allons associer ce service à ce type de patron. Il serait même intéressant de détecter si certains paramètres d'appels sont en corrélation avec certains chemins de sortie spécifiques. La figure 3.10(e) montre la carte de règles spécifiée pour le patron Routeur.

1 RULE_CARD : ***Basic Service*** {
2 RULE : ***Basic Service*** {INTER ***HighSR HighCOH HighA LowRT***};
3 RULE : ***HighSR*** {SR $\geq$ HIGH};
4 RULE : ***HighCOH*** {COH $\geq$ HIGH};
5 RULE : ***HighA*** {A $\geq$ HIGH};
6 RULE : ***LowRT*** {RT $\leq$ LOW};
7 };

(a) Patron optimal

1 RULE_CARD : ***Facade*** {
2 RULE : ***Facade*** {INTER ***HighDR LowIOCR HighRT***};
3 RULE : ***HighDR*** {DR $\geq$ HIGH};
4 RULE : ***LowIOCR*** {NIC/NOC $\leq$ LOW};
5 RULE : ***HighRT*** {RT $\geq$ HIGH};
6 };

(b) Patron Façade

1 RULE_CARD : ***Proxy*** {
2 RULE : ***Proxy*** {INTER ***EqualIOCR HighSR LowPSC***};
3 RULE : ***EqualIOCR*** {NIC/NOC = 1.0};
4 RULE : ***HighSR*** {SR $\geq$ HIGH};
5 RULE : ***LowPSC*** {PSC $\leq$ LOW};
6 };

(c) Patron Proxy

1 RULE_CARD : ***Adapter*** {
2 RULE : ***Adapter*** {INTER ***EqualIOCR HighPSC***};
3 RULE : ***EqualIOCR*** {NIC/NOC = 1.0};
4 RULE : ***HighPSC*** {PSC $\geq$ HIGH};
5 };

(d) Patron Adaptateur

1 RULE_CARD : ***Router*** {
2 RULE : ***Router*** {***HighPOPC***};
3 RULE : ***HighPOPC*** {POPC $\geq$ HIGH};
4 };

(e) Patron Routeur

Figure 3.10 Cartes de règles des patrons SOA.

CHAPITRE IV

IMPLÉMENTATION DE SODOP

Afin d'adapter notre approche de détection aux patrons SOA, nous avons dû remanier le code existant de la plateforme SOFA (*Service Oriented Framework for Analysis*). L'implémentation réalisée initialement pour la détection d'anti-patrons de l'approche SODA n'était pas du tout modulaire. En effet, elle était fortement liée au processus de détection de mauvaises pratiques ; nous l'avons désormais étendue au concept de détection de motifs. Afin de satisfaire les deux approches SODA et SODOP, nous décrivons un motif au sens générique comme pouvant faire référence à la fois à un patron et un anti-patron. La plateforme comportait beaucoup d'incohérences et ne permettait pas l'ajout aisé de nouvelles métriques sans modifier les métriques existantes. Auparavant, chaque carte de règles était implémentée de façon statique et chaque calcul de métrique était redondant, ce qui nuisait gravement à la performance de l'outil. Nous avons entrepris une refactorisation du code, en particulier pour obtenir une implémentation de chaque règle et de chaque métrique beaucoup plus générique et automatisable. Dans les sections suivantes, nous commençons par rappeler le fonctionnement global de la plateforme SOFA pour détailler les changements majeurs réalisés sur certains modules.

4.1 Description de la plateforme SOFA

La plateforme SOFA (*Service Oriented Framework for Analysis*) regroupe l'ensemble des modules permettant de concrétiser l'approche SODOP proposée dans ce

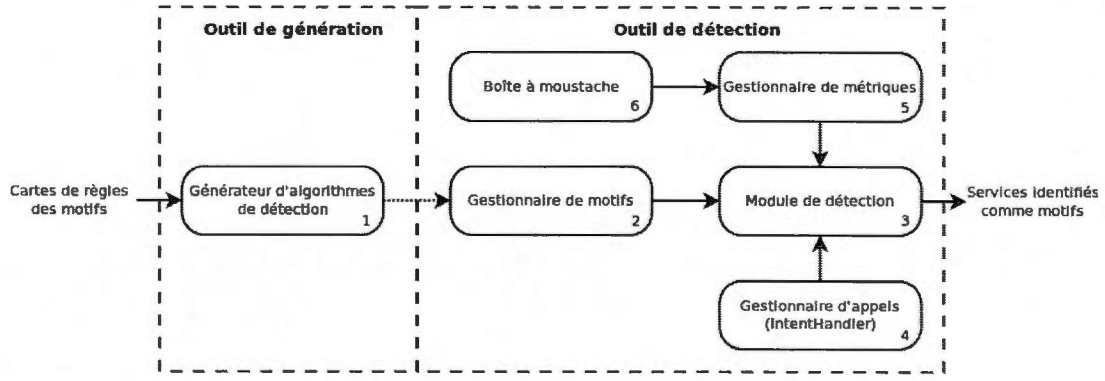

Figure 4.1 Modules de la plateforme SOFA.

mémoire. Celle-ci est actuellement décomposée en six modules principaux, les objectifs de chacun étant les suivants :

Module 1 *: Le générateur d'algorithmes de détection* est le module permettant de transformer la spécification formelle des cartes de règles en classes de détection de motifs.

Module 2 *: Le gestionnaire de motifs* a pour but de traiter les motifs à détecter et surtout d'extraire les métriques nécessaires à la détection.

Module 3 *: Le module de détection* est le cœur de la plateforme, il coordonne tout le processus de détection en communiquant avec les autres modules. Il expose les différents services détectés en tant que motifs à la fin du processus.

Module 4 *: Le gestionnaire d'appels* joue le rôle d'observateur, il intervient à chaque fois qu'un appel de service est réalisé et délègue les tâches de calculs au module 5.

Module 5 *: Le gestionnaire de métriques* coordonne les calculs de chaque métrique et stocke les résultats dans le module 6.

Module 6 *: La boîte à moustache* restitue les services présents dans chaque intervalle en fonction des valeurs ajoutées durant l'exécution.

L'architecture actuelle de la plateforme SOFA nous permet une plus grande modularité de chaque composant. De ce fait, le traitement de nouvelles cartes de règles et l'intégration de nouvelles métriques sont beaucoup plus transparents, et ils induisent moins de complexité.

4.2 Implémentation des cartes de règles

Nos travaux ont été menés conjointement avec ceux de Christopher Robert, un stagiaire de l'eXia.Cesi ayant travaillé sur la génération automatique de cartes de règles en algorithmes (module 1 de la plateforme SOFA). La modélisation du DSL a été réalisée à l'aide du modèle EMF (*Eclipse Modeling Framework*) provenant de l'outil de développement intégré *Eclipse* (Eclipse Modeling Framework Project, 2011). L'écriture des cartes de règles ainsi que les tâches d'analyse grammaticale du langage du DSL ont été faites via *EMFText* (Eclipse Modeling Framework Text Plugin, 2013), une extension permettant l'intégration d'un modèle *EMF*. L'outil de génération de code *Acceleo* (Acceleo Code Generator Tool, 2010) a permis de générer les algorithmes de détection, sous forme d'une classe Java par carte de règles, en respectant le modèle attendu par le module de détection.

Le remaniement complet de la plateforme a nécessité une modélisation minutieuse en fonction de la grammaire désormais fixe de notre DSL. Nous avons été confrontés à un premier problème, qui n'avait pas été pris en compte dans la précédente version : le fait qu'une carte soit définie comme une combinaison d'une ou plusieurs règles mais aussi, et surtout, potentiellement composée d'une carte de règles existante. Nous avons donc modélisé la combinaison de règles comme un composite pour pouvoir représenter une structure arborescente. L'arborescence retenue est composée de feuilles (règles) ou composites (ensemble de plusieurs feuilles ou composites). Chaque patron (classe spécialisée de *AMotif*) sera donc associé au motif le plus abstrait (*Smell*) comme le montre la figure 4.2.

Conformément à la grammaire du DSL, chaque feuille (*SmellLeaf*) de la hiérarchie

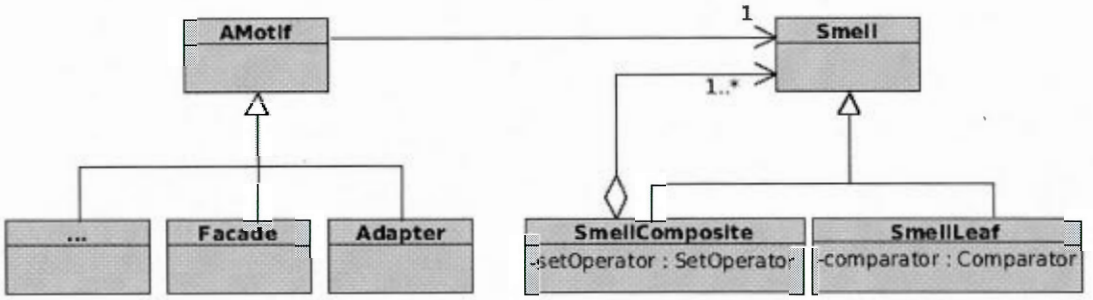

Figure 4.2 Structure des cartes de règles suivant le patron composite.

pourra alors représenter une des différentes règles de la carte. Celles-ci peuvent cependant être de trois types différents suivant la comparaison réalisée avec les opérateurs de comparaison (*Comparator*) :

1. Une métrique comparable à une valeur numérique (*SmellNumericValue*).
2. Une métrique comparable à une valeur ordinale (*SmellOrdinalValue*).
3. Une métrique comparable à une autre métrique (*SmellMetricValue*).

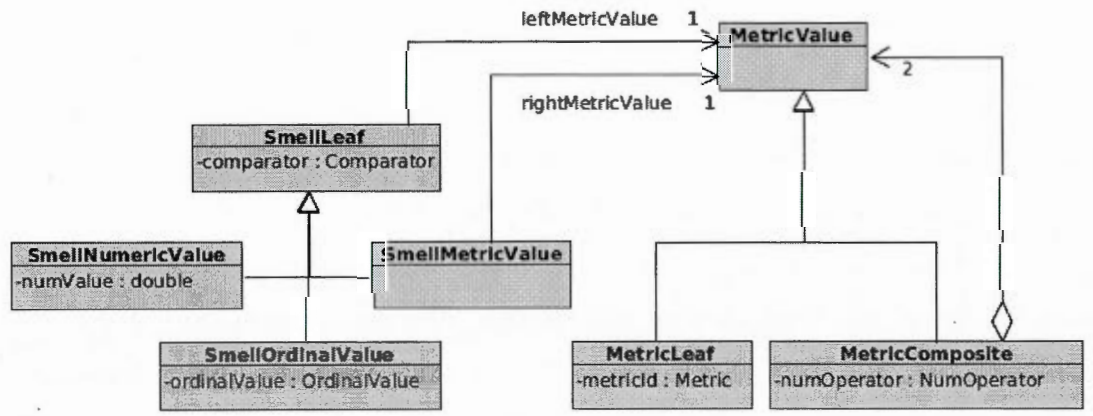

Figure 4.3 Structure des métriques présentes dans chaque règle.

Il est aussi possible de définir une composition de métriques existantes (*MetricComposite*). C'est pourquoi, une autre architecture en composite a été retenue ici,

comme présentée sur la figure 4.3. Les options de comparateur, opérateur, valeurs ordinales et métriques ont été modélisées à l'aide d'énumérations, comme décrit sur le figure 4.4. Celles-ci sont stables et n'apportent aucun nouveau comportement, elles ne nécessitent donc pas de classe à part entière.

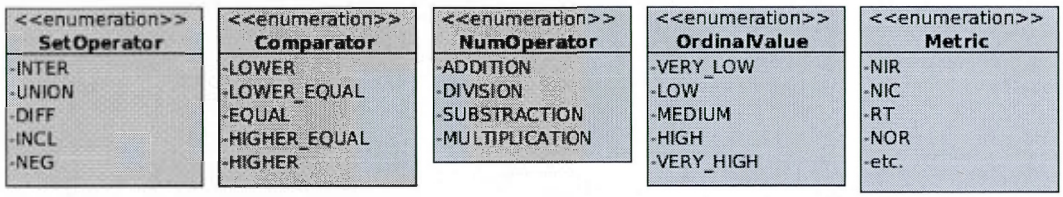

Figure 4.4 Valeurs d'énumérations possibles.

Afin de mieux montrer comment chaque instance de notre architecture se comporte à l'exécution, un exemple concret est décrit ci-dessous. Nous prenons l'exemple d'une carte de règles simple : le ratio des métriques NIC et NOC doit être égal à 1.0, tel que spécifié sur la figure 4.5. Cette option aurait aussi pu être représentée avec l'égalité directe des deux valeurs de métriques. Dans notre cas, la règle est de type valeur numérique (*SmellNumericValue*) avec la valeur de 1.0 et le comparateur d'égalité. On représente le ratio de deux métriques via la classe *MetricComposite* avec l'opérateur *DIVISION*, composé de deux feuilles associées respectivement aux métriques *NIC* et *NOC*.

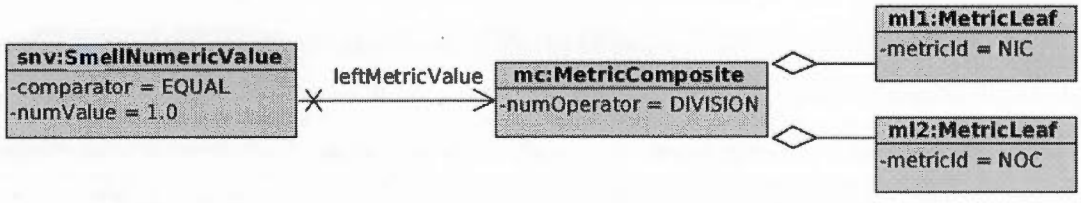

Figure 4.5 Exemple de diagramme objet pour la règle $\frac{NIC}{NOC} = 1.0$.

Prenons l'exemple du patron *Routeur* afin d'expliquer l'implémentation des motifs utilisée dans la plateforme SOFA. Sa carte de règles présentée sur la figure 3.10(e) (page 34) comporte une seule règle. Cette règle a pour but de retenir uniquement les services ayant une proportion de changement de chemins de sortie (POPC) élevée ou très élevée. L'implémentation présentée sur la figure 4.6 détaille l'implémentation de cette carte de règles dans la plateforme SOFA. La classe *Router* représente la définition d'un nouveau patron en héritant de *AMotif* (ligne 9). La métrique *POPC* déclarée à la ligne 13 est assignée (ligne 17) à l'unique règle du patron (ligne 14) en spécifiant qu'elle ne retient que les valeurs ordinales hautes ou très hautes.

```
1  package com.sodop.patterns;
2
3⊕ import com.sofa.metric.Metric;
8
9  public class Router extends AMotif {
10
11⊖    public Router() {
12        // SMELL 1
13        MetricValue metricValue1Smell1 = new MetricLeaf(Metric.POPC);
14        Smell smell1 = new SmellOrdinalValue(metricValue1Smell1, Comparator.GREATER, OrdinalValue.HIGH);
15
16        // ROOT SMELL
17        this.rootSmell = smell1;
18    }
19 }
```

Figure 4.6 Implémentation du patron Routeur dans SOFA.

4.3 Détection et calcul des métriques

Nos expérimentations se basent sur un processus de détection réalisé durant l'exécution d'un système. Nous tirons parti des fonctionnalités présentes dans l'implémentation du standard SCA réalisée par la plateforme *FraSCAti* afin d'intercepter l'ensemble des communications inter et intra-composants. En effet, *FraSCAti* fournit la possibilité d'attacher un gestionnaire (*IntentHandler*) sur chaque composant d'un système. Dès qu'un service du composant observé est appelé, le gestionnaire est automatiquement notifié de l'appel et peut ainsi effectuer des pré et/ou post-traitements sur celui-ci. Il est alors possible de récupérer le nom des composants émetteur et destinataire, le nom

de la méthode invoquée ainsi que les arguments utilisés pour cet appel. Il faut noter que les post-traitements d'un appel ne sont pas réalisés tant que les appels imbriqués ne sont pas encore terminés. De cette façon, nous avons pu reconstituer la pile d'appels d'un scénario complet d'exécution pour les systèmes étudiés. La figure 3.5 (page 28) représente un exemple de scénario avec la pile d'exécution associée, afin de montrer les dépendances entre composants.

Dans notre cas, nous avons tissé des points de jointure (*join points*) sur chaque service dans le but de générer un évènement pour chaque appel. Chaque évènement est intercepté, pour ensuite calculer les métriques nécessaires sur le service en question.

Afin de pouvoir calculer les valeurs des différentes métriques durant l'exécution même d'un système, nous avons basé notre processus de détection sur le gestionnaire d'événements. En prenant compte de la liste des patrons SOA à détecter sur le système, la première étape consiste à référencer la liste des métriques à calculer. Le tableau 4.1 détaille les métriques actuellement disponibles dans l'outil, celles-ci pouvant être combinées pour former des cartes de règles. Nous ne souhaitons bien sûr pas calculer toutes les métriques pour chaque exécution afin de minimiser les temps de détection de notre outil. Ensuite pour chaque appel d'un des services du système, un événement est levé et capturé par le gestionnaire. Celui-ci stocke les informations relatives à cet appel dans un arbre visant à rassembler l'ensemble des informations du scénario d'exécution. Les métriques statiques de notre outil sont directement calculées au premier appel du service. Les métriques dynamiques, quant à elles, sont calculées à la fin de l'exécution, en récupérant les informations nécessaires dans l'arbre d'appels.

La figure 4.7 montre le comportement du gestionnaire (*IntentHandler*), module 4 de la plateforme SOFA, pour plusieurs appels de méthodes (*m1, m2*) sur le service *ServiceA*. Le tout premier appel induit le calcul de la métrique statique *NIR* qui engendre l'ajout de la valeur dans la boîte à moustache correspondant à cette métrique (Boxplot NIR). On voit sur le deuxième appel de méthode que cette métrique n'est plus calculée puisqu'elle est statique. En revanche, l'ajout des appels sur l'arbre d'appels est

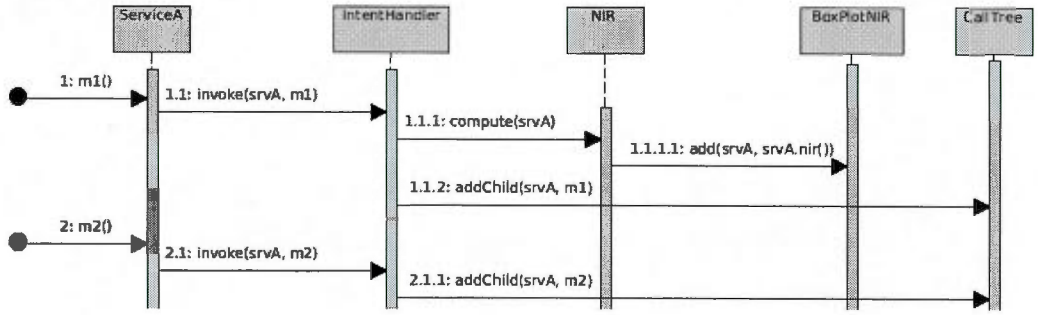

Figure 4.7 Comportement du gestionnaire pour les appels de services.

toujours réalisé (*CallTree*) dans le but de calculer les métriques dynamiques à la fin de l'exécution.

Nom	Type	Description anglaise	Description française
A	(D)	Availability	Disponibilité
ANAM	(S)	Average Number of Accessor Methods	Nombre moyen d'accesseurs
ANIM	(S)	Average Number of Identical Methods	Nombre moyen de méthodes identiques
ANP	(S)	Average Number of Parameters	Nombre moyen de paramètres
ANPT	(S)	Average Number of Primitive Type	Nombre moyen de paramètres primitifs
COH	(S)	COHesion	Cohésion
DR	(D)	Delegation Ratio	Ratio de délégation
ET	(D)	Execution Time	Temps d'exécution
NDC	(D)	Number of Different Clients	Nombre de clients différents
NIC	(D)	Number of Incoming Calls	Nombre d'appels entrants
NID	(S)	Number of Interfaces Declared	Nombre d'interfaces déclarées
NIR	(S)	Number of Incoming References	Nombre de références entrantes
NMD	(S)	Number of Methods Declared	Nombre de méthodes déclarées
NOC	(D)	Number of Outgoing Calls	Nombre d'appels sortants
NOR	(S)	Number of Outgoing References	Nombre de références sortantes
NPRIMD	(S)	Number of Private Methods Declared	Nombre de méthodes privées déclarées
NPROMD	(S)	Number of Protected Methods Declared	Nombre de méthodes protégées déclarées
NPUMD	(S)	Number of Public Methods Declared	Nombre de méthodes publiques déclarées
NSC	(S)	Number of Sub Components	Nombre de sous-composants
NTMI	(D)	Number of Transitive Methods Invoked	Nombre de méthodes transitives invoquées
POPC	(D)	Proportion of Outgoing Paths Change	Proportion de chemins sortants différents
PSC	(D)	Proportion of Signatures Change	Proportion de changements de signature
RT	(D)	Response Time	Temps de réponse
SR	(D)	Service Reuse (NIC / TotalCalls)	Taux de réutilisation
TNP	(S)	Total Number of Parameters	Nombre total de paramètres

Tableau 4.1 Liste des métriques de SODOP (S : Statique, D : Dynamique).

CHAPITRE V

EXPÉRIMENTATIONS ET RÉSULTATS

Afin de montrer la pertinence de l'approche SODOP, nous avons réalisé diverses expérimentations, qui sont présentées dans ce chapitre. Nous avons commencé par spécifier les cinq patrons SOA décrits dans le chapitre 3.2 sous forme de cartes de règles comme présenté à la figure 3.10. Nous avons retenu deux systèmes SCA en tant que candidats à la détection de ces patrons, *Home-Automation* et *FraSCAti* (FraSCAti Open-Source SCA Implementation, 2009). *Home-Automation* est un système qui permet de réaliser des opérations de domotique, alors que *FraSCAti* est une implémentation du standard SCA. Les expérimentations visent à montrer (i) la flexibilité du DSL pour la spécification de patrons SOA, (ii) l'efficacité des algorithmes de détection et enfin, (iii) la précision de la plateforme sous-jacente. Dans le cadre de nos expérimentations, deux analystes indépendants ont validé les résultats obtenus sur *Home-Automation*. L'équipe même de développement de FraSCAti a pu valider les résultats obtenus de la détection de patrons SOA sur leur plateforme. Ces validations indépendantes nous permettent de calculer la précision et le rappel de SODOP. De hautes valeurs de précision et rappel permettent de démontrer la pertinence et l'efficacité de la spécification de patrons SOA ainsi que des algorithmes de détection associés. Nous commençons par présenter plus en détail les systèmes analysés, les processus de détection ainsi que les processus de validation utilisés. Nous décrivons ensuite les résultats obtenus lors de nos expérimentations pour enfin les analyser et soutenir notre approche.

5.1 Hypothèses

Les expérimentations menées ont pour but de valider les trois hypothèses suivantes :

H1. Extensibilité : *Le DSL étendu proposé est assez flexible pour la spécification de patrons SOA.* Par cette hypothèse, nous cherchons à démontrer que même si le DSL et la plateforme initiale étaient dédiés à la spécification et à la détection d'anti-patrons, ils sont désormais assez modulaires pour prendre aussi en compte les bonnes pratiques de conception SOA via l'utilisation de nouvelles métriques.

H2. Exactitude : *Les services identifiés comme patrons SOA dans nos expérimentations doivent atteindre un minimum de 80% de précision et 100% de rappel.* En effet, nous voulons garantir la précision et l'efficacité des cartes de règles et algorithmes de détection associés. Nous voulons identifier tous les patrons présents dans les systèmes analysés tout en évitant au maximum les faux positifs avec une haute précision.

H3. Performance : *Le temps nécessaire à la détection des patrons via les algorithmes ne doit pas impacter la performance et le temps d'exécution du système analysé.* Nous voulons garder le temps de détection requis par SODOP et la plateforme SOFA au plus bas pour éviter des ralentissements d'exécution.

5.2 Systèmes analysés et processus de détection

Les expérimentations ont été menées sur deux systèmes à architecture orientée composants qui suivent les principes des architectures orientées services. *Home-Automation* est un système permettant de manipuler des appareils électroniques à distance pour réaliser de la domotique. Ce système fait partie des applications de démonstration disponibles dans la plateforme *FraSCAti* (FraSCAti Open-Source SCA Implementation, 2009). Il a notamment été conçu pour assister et faciliter les personnes âgées dans leurs tâches quotidiennes. Il est formé de 13 composants exposant chacun un unique service, comme le montre la figure 5.1. Notre simulation d'exécution utilise

sept scénarios différents afin de représenter une couverture de détection la plus large possible. *FraSCAti* est une implémentation libre du standard SCA, elle-même conçue suivant ce type d'architecture orientée composants (FraSCAti Open-Source SCA Implementation, 2009). C'est la plus importante application actuellement disponible de cette plateforme, puisqu'elle expose 13 composants externes (composites), qui encapsulent plusieurs autres composants pour une évaluation totale de 91 composants. Afin d'effectuer notre détection sur la plateforme *FraSCAti* elle-même, nous avons simulé l'amorçage et l'exécution de six applications différentes.

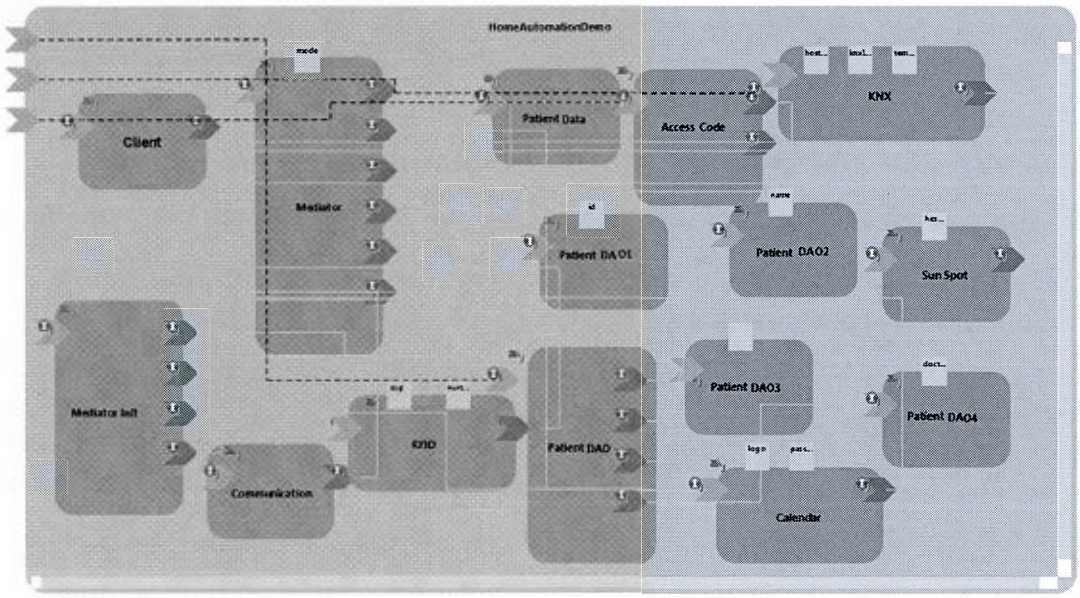

Figure 5.1 Architecture du système *Home-Automation*.

Le processus utilisé pour la détection des patrons SOA dans les applications SCA suivent les trois étapes de l'approche SODOP présentée au chapitre 3. Les cinq patrons SOA décrits précédemment sont, dans un premier temps, spécifiés via les cartes de règles du DSL. Ensuite, nous générons les algorithmes de détection correspondants dans la seconde étape. Enfin, nous les appliquons sur les deux systèmes *Home-Automation* et

FraSCAti pour y détecter les potentiels candidats aux patrons spécifiés. Nous validons nos résultats en calculant la précision — la proportion de réels patrons parmi ceux détectés — et le rappel — la proportion de patrons détectés sur l'ensemble des patrons existants dans l'application. Ces validations ont été réalisées via une analyse statique et manuelle de chaque composant des deux systèmes. Les calculs ont été réalisés par deux ingénieurs logiciels externes de notre équipe avec une bonne expertise en conception de systèmes SOA. Ceux-ci nous permettent d'assurer la cohérence des résultats sans les biaiser pour autant. Nous avons aussi eu le retour de l'équipe de développement de FraSCAti afin de valider les résultats.

5.3 Résultats obtenus

Les deux sections qui suivent présentent en détails les résultats de la détection obtenus sur les deux systèmes, SOA *Home-Automation* et *FraSCAti*.

5.3.1 Détails des résultats sur *Home-Automation*

Quatre des cinq patrons SOA spécifiés ont été détectés sur *Home-Automation* — le patron Adaptateur n'a pas été détecté comme le montre le tableau 5.1. Les composants *patientDAO*, *communication* et *knxMock* ont été détectés comme des services optimaux. Ils disposent en effet de valeurs de cohésion fortes (COH $\geq$ 0.34 pour une médiane de 0.21), de haute réutilisabilité (SR $\geq$ 0.10 pour une médiane de 0.06) et de temps de réponse très faibles (RT $\leq$ 0.25ms correspondant à des valeurs inférieures au premier quartile). Conformément à la définition du patron Service Optimal, ces trois composants représentent donc les services du système qui sont les mieux conçus car ils respectent les principes de base de conception logicielle. Le composant *mediator* est à la fois considéré comme Façade et Routeur. La façade représente un service jouant le rôle de couche de présentation pour les clients afin de cacher la complexité du sous-système. Sa proportion de délégation (DR = 1) montre effectivement qu'il délègue l'ensemble de ses appels entrants. De plus, ses appels sortants sont presque six fois plus nombreux

Tableau 5.1 Résultats de la détection de patrons SOA sur *Home-Automation*.

Patron SOA	Composants détectés	Métriques	Temps de détec.	Temps d'exéc.	Précision	Rappel
Service optimal	*patientDAO* *communication* *knxMock*	COH RT SR 0.49 0.25ms 0.24 0.34 0.24ms 0.10 0.38 0.16ms 0.11	80ms	6.82s	[3/2] 66.6%	[2/2] 100%
Façade	*mediator*	NIC/NOC DR RT 0.17 1.0 2.8ms	10ms	6.66s	[1/1] 100%	[1/1] 100%
Proxy	*patientDAO*	NIC/NOC SR PSC 1.0 0.24 0.0	13ms	6.74s	[1/1] 100%	[1/1] 100%
Adapteur	n/a	n/a	10ms	6.76s	[0/0] 100%	[0/0] 100%
Routeur	*mediator*	POPC 0.5	11ms	6.67s	[1/1] 100%	[1/1] 100%
MOYENNE			25ms	6.73s	93.3%	100%

pour un seul appel entrant (NIC/NOC = 0.17), ce qui traduit un haut temps de réponse (RT = 2.8ms). Le *mediator* a aussi été détecté en tant que routeur du fait de sa haute valeur de changement de chemins de sortie (POPC = 0.5). Cette valeur caractérise le *mediator* comme un composant distribuant la moitié de ses appels entrants sur des appels sortants différents. Le composant *patientDAO* correspond également au patron Proxy du fait de sa haute réutilisation (SR = 0.24 comparé à une médiane de 0.06), et aussi pour le relais systématique de ses appels (NIC = NOC) avec les mêmes signatures de méthode (PSC = 0).

Nous pouvons observer que le temps requis pour la détection de chaque patron est en moyenne de 25ms, alors que le le temps moyen d'exécution du système sur un ensemble de scénarios donné est de 6.73s. Ces valeurs démontrent donc le faible impact

de la détection de patrons sur l'exécution du système, et donc sur les résultats. Pour finir, la validation réalisée par les deux experts externes ont mené à une précision de 93.3% et un rappel de 100%, ce qui indique que tous les patrons existant dans *Home-Automation* ont été détectés et que peu de faux positifs ont été rapportés.

5.3.2 Détails des résultats sur *FraSCAti*

Tableau 5.2 Résultats de la détection de patrons SOA sur *FraSCAti*.

Patron SOA	Composants détectés	Métriques			Temps de détec.	Temps d'exéc.	Précision	Rappel
Service optimal		COH	RT	SR	241ms	11.34s	[5/5] 100%	n/a
	sca-interface	0.48	0.09ms	0.11				
	sca-interface-java	0.50	0.02ms	0.11				
	sca-impl.	0.48	0.67ms	0.05				
	sca-impl.-java	0.50	0.59ms	0.04				
	sca-comp.-service	0.48	0.25ms	0.48				
Façade		NIC/NOC	DR	RT	57ms	10.62s	[3/3] 100%	[3/16] 18.7%
	FraSCAti	0.46	1.0	571ms				
	assembly-factory	0.25	1.0	181ms				
	composite-parser	0.63	1.0	10ms				
Proxy		NIC/NOC	SR	PSC	65ms	10.72s	[3/3] 100%	[3/14] 21.4%
	sca-interface	1.0	0.11	0.0				
	sca-impl.	1.0	0.05	0.0				
	component-factory	1.0	0.11	0.0				
Adapteur	*BindingFactory*	NIC/NOC	PSC		57ms	10.96s	[1/1] 100%	[1/14] 7.1%
		1.0	1.0					
Routeur	n/a	n/a			67ms	10.88s	[0/0] 100%	[0/7] 0.0%
MOYENNE					97ms	10.90s	100%	11.8%

Le tableau 5.2 montre que la détection des patrons SOA sur *FraSCAti* donne plus de résultats que sur *Home-Automation* — plus de composants sont détectés en tant que patrons. Ces résultats nombreux sont en partie expliqués par la taille du système *FraS-*

CAti qui représente tout de même plus de 10 fois *Home-Automation* en termes de composants. Cinq composants ont été détectés en tant que Service Optimal, car ils sont très réutilisés (SR > 0.1 comparé à la médiane de 0.003), très cohésifs (COH > 0.48) et ont un temps de réponse très bas (RT < 0.7ms). Les composants majeurs de la plateforme, *FraSCAti*, *assembly-factory* et *composite-parser*, sont détectés comme des façades car ils sont des points d'entrée du système et relaient l'ensemble des appels entrants (DR = 1). Leur ratio d'appels entrants/sortants (NIC/NOC = 0.46, 0.25 et 0.63) restent bas comparés à la médiane de 1.0. Ils jouent aussi le rôle de façade car ils ont les temps de réponse parmi les plus élevés (respectivement de 571ms, 181ms et 10ms), essentiellement à cause de leurs appels sortants très nombreux. Le patron Proxy est détecté sur les trois composants suivants : *sca-interface*, *sca-implementation* et *component-factory*. Ils ont été identifiés comme Proxy car ils représentent des services relais (NIC/NOC = 1) très réutilisés (SR > 0.5) qui empruntent les mêmes signatures de méthodes entrantes et sortantes (PSC = 0). Le seul patron SOA manquant dans *Home-Automation* et découvert dans *FraSCAti* est l'adaptateur sur le composant *BindingFactory*. Il joue ce rôle car il relaie tous ses appels entrants (NIC = NOC) mais surtout parce qu'il adapte les signatures de méthodes des appels (PSC = 1, qui indique une forte proportion de changement). Enfin, à la différence de *Home-Automation*, aucun routeur n'a été détecté dans *FraSCAti*.

Le temps moyen nécessaire à nos expérimentations est de 97ms pour une durée d'exécution totale de 10.9s. De façon similaire à *Home-Automation*, la détection représente moins de 1% du temps d'exécution total, et n'affecte donc pas la performance, même avec un système aussi grand. Nous avons soumis les résultats de détection de patrons aux membres de l'équipe de développement de *FraSCAti*. Ils ont confirmé les résultats de la détection sur tous les composants. Cependant, le rappel n'est que de 11.8%. Nos algorithmes de détection n'ont donc pas réussi à détecter l'ensemble des patrons SOA présents dans *FraSCAti*. Ceci s'explique en grande partie par le fait que les scénarios utilisés ne couvrent pas tous les chemins d'exécution possibles.

5.4 Analyse des résultats

Nous analysons les résultats afin de vérifier les hypothèses énoncées au début de l'étude. Celles-ci nous permettront de savoir si l'approche SODOP proposée apporte un réel gain dans une démarche d'assurance qualité des systèmes à base de services.

5.4.1 Discussion

H1. Extensibilité : *Le DSL étendu proposé est assez flexible pour la spécification de patrons SOA.* Cette première hypothèse est confirmée car nous avons montré au travers des expérimentations que le DSL étendu proposé répond aux nouvelles attentes. En effet, il permet aux concepteurs de définir différents types de cartes de règles en ajoutant même de nouvelles métriques, à la fois statiques et dynamiques. Dans notre cas, la spécification des patrons SOA a nécessité l'ajout de huit nouvelles métriques tout en réutilisant les 14 existantes. En plus de cet ajout de métriques, le DSL a été grandement simplifié ; nous y avons également ajouté la possibilité de combiner plusieurs métriques à l'aide d'opérateurs numériques (+,-,*,/). Cette fonctionnalité nous a permis d'éviter l'introduction de nouvelles métriques inutiles pour garder le langage aussi simple et flexible que possible.

H2. Exactitude : *Les services identifiés comme patrons SOA dans nos expérimentations doivent atteindre un minimum de 80% de précision et 100% de rappel.* Les résultats de la détection montrent l'efficacité de l'approche SODOP avec respectivement 93% et 100% de précision pour *Home-Automation* et *FraSCAti*. Le rappel pour *Home-Automation* est de 100% mais celui de *FraSCAti* est seulement de 12% environ. Ce résultat est directement lié à la nature hautement dynamique de la détection de patrons, qui est basée sur un ensemble de scénarios qui ne couvrent pas tous les chemins d'exécution possibles du système. Dans ces expérimentations avec *FraSCAti*, et contrairement à *Home-Automation*, il est très difficile d'atteindre une couverture maximale compte tenu de la taille du système et des différentes technologies supportées.

H3. Performance : *Le temps nécessaire à la détection des patrons via les algorithmes ne doit pas impacter la performance et le temps d'exécution du système analysé.* Comme illustré dans les tableaux 5.1 et 5.2, peu importe le nombre de patrons SOA à détecter, et donc le nombre de métriques à calculer, la durée de détection reste faible par rapport à la durée d'exécution totale. Cela n'influence donc pas négativement sur la performance du système. Nous avons fait le constat que la seule propriété pouvant induire une augmentation du temps de détection serait le nombre de services impliqués, car les métriques sont calculées sur chacun d'eux. *FraSCAti* compte presque huit fois plus de composants que *Home-Automation*, ce qui explique alors le temps proportionnel requis pour le calcul des métriques (environ 1% du temps total d'exécution). Du fait que ces expérimentations aient été exécutées localement, les temps d'exécution sont intimement liés à la puissance computationnelle de la machine. Dans ces expérimentations, nous avons utilisé un processeur Intel E5345 avec 4GB de RAM.

5.4.2 Menaces à la validité

Diverses menaces peuvent affecter la validité de notre étude. Premièrement, la fiabilité externe — c'est-à-dire la reproduction de nos expérimentations — est seulement garantie à condition que les mêmes capacités computationnelles soient utilisées. Celle-ci est aussi garantie par le fait que les algorithmes de détection soient générés automatiquement suivant des motifs de base prédéfinis qui resteront les mêmes pour une même carte de règles. Nous proposons les détails de nos résultats et des systèmes analysés sur le site internet SOFA (`http://sofa.uqam.ca/sodop`).

La principale menace à la validité externe de notre étude peut provenir du fait que nous n'utilisons que des technologies SCA. Bien qu'elles soient représentatives à la fois de petits et de grands systèmes, les technologies SOA ont souvent des caractéristiques spécifiques, c'est pourquoi nous planifions d'étendre nos expérimentations, dans le futur, à d'autres technologies comme les services Web et les services REST.

Nous avons tenté de minimiser le plus possible les menaces à la validité de

construction de notre étude en fournissant des jeux de scénarios les plus complets et représentatifs possibles pour chaque système étudié. Cependant, pour *FraSCAti*, les scénarios n'étaient visiblement pas assez complets comme le montre le rappel de 11.8%. Du fait de la taille du système, nous allons étendre au maximum ces scénarios dans les expérimentations à venir.

L'autre menace possible à la validité de construction a pour origine la subjectivité des cartes de règles définies. Cette spécification dépend en effet grandement du concepteur qui les définit. Nous avons toutefois essayé dans la mesure du possible de rester le plus proche et fidèle à la description de la littérature. De plus, bien que nous n'ayons défini que cinq patrons SOA sous la forme de cartes de règles dans notre approche SODOP, ils sont représentatifs de la littérature. Même si les catalogues SOA se concentrent souvent sur les patrons propres à une technologie particulière, nous avons essayé de spécifier des patrons SOA génériques et expressifs.

CONCLUSION

Les patrons SOA sont de bonnes pratiques reconnues pour résoudre des problèmes récurrents et communs lors de la conception de systèmes à base de services. Notre approche en trois étapes, appelée SODOP (*Service Oriented Detection Of Patterns*), consiste en la spécification et la détection de patrons SOA pour atteindre de bonnes propriétés de conception et de qualité de service sur des systèmes SOA. La première étape consiste à spécifier une carte de règles — un ensemble de règles combinant des métriques statiques et dynamiques — pour chaque patron. Cinq patrons SOA ont été définis dans notre étude, impliquant 22 métriques statiques et dynamiques différentes, dont huit métriques dynamiques nouvellement intégrées. La seconde étape consiste à générer automatiquement des algorithmes de détection pour chaque carte de règles. Et enfin, la troisième étape consiste à appliquer le processus de détection complet sur des systèmes SOA. Nous avons validé notre approche sur deux systèmes SCA : *Home-Automation*, une application avec 13 services pour réaliser des opérations de domotique, et *FraSCAti*, une implémentation du standard SCA fournissant 91 composants. Nous avons montré à travers les expérimentations que nous pouvons obtenir de hautes valeurs de précision et de rappel à condition que les scénarios d'exécution soient exhaustifs.

Plusieurs pistes de recherche sont projetées ou présentement explorées par notre groupe de recherche. Premièrement, nous allons étendre l'approche SODOP par la spécification d'autres patrons SOA, en les appliquant sur un groupe de systèmes plus important et représentatif. Nous planifions par exemple d'appliquer notre approche sur d'autres technologies SOA, notamment les services Web et les services REST, car ils partagent plusieurs caractéristiques communes. Cependant, notre approche reste pour le moment applicable si nous enveloppons les services d'autres technologies à l'intérieur même de conteneurs SCA.

APPENDICE A

ARTICLE *DETECTION OF SOA PATTERNS* PUBLIÉ DANS LA CONFÉRENCE ICSOC 2013

Detection of SOA Patterns

Anthony Demange, Naouel Moha, and Guy Tremblay

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

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

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

1 Introduction

Service Oriented Architecture (SOA) is an architectural style increasingly adopted because it offers system architects a high level solution to software design. SOA systems are based upon loosely coupled, autonomous and reusable coarse-grained components called services [22]. Each service provides a domain specific behavior, and services can be composed as composite to fulfill high level business processes requirements. Various technologies have emerged to implement this style, among them, Web Services [14] and SCA [6]. Google, Amazon, Microsoft are well-known businesses that have successfully based their information systems on SOA. Software systems evolve rapidly due to the addition of new functionalities and the integration of legacy systems. Well designed systems tend to reduce maintenance effort and costs in the long term [4,21]. However, designing

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

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

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

In this paper, we extend the existing SOFA framework to consider the detection of SOA patterns at runtime. Until now, no automatic approach for the detection of such patterns has been proposed, making the approach proposed in this paper original. The proposed approach is called SODOP (Service Oriented Detection Of Patterns) and consists in the following three contributions. (1) A thorough domain analysis from different catalogs led us to compile and categorize the best practices in SOA systems and their underlying technologies. (2) This analysis resulted in the specification of five significant SOA patterns using rule cards. We selected these five SOA patterns because they represent technology-agnostic, common and recurrent good quality practices in the design and QoS of SOA systems. (3) Specifying the appropriate rule cards required us to extend SOFA's existing set of metrics with eight new metrics. We validated

the proposed approach with two SOA systems: *Home-Automation*, a system that provides services for domotic tasks, and *FraSCAti*, an implementation of the *Service Component Architecture* (SCA) standard [24]. We show that our SODOP approach allows the specification and detection of SOA patterns with high precision values. More detailed information on our approach and the analyzed systems can be found through the SOFA website (`http://sofa.uqam.ca/sodop`).

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

2 Related Work

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

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

The majority of the community tends to say SOA was first introduced in 1996 by Schulte and Natiz in their Gartner technical report [23]. SOA patterns catalogs only appeared starting around 2009 [7,9,22]. Galster *et al.* identified most of the patterns specified in Erl's *SOA Patterns* [9] and showed the positive impacts of patterns on quality attributes [11]. Their approach, manual, consists in spec-

ifying quality attributes on each pattern and then identifying them manually in real service-based systems.

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

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

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

3 Our Approach SODOP

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

3.1 About the Service Component Architecture

Before introducing the SODA and SODOP approaches, it must be stressed that the following experiments were made with the Service Component Architecture

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

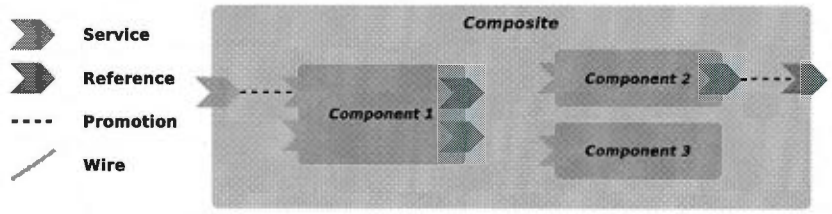

Fig. 1. Key Concepts of the SCA Standard.

3.2 Description of the Earlier SODA Approach

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

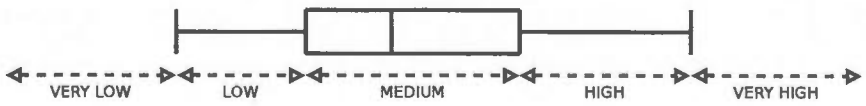

Fig. 2. Mapping between ordinal values and box-plot intervals.

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

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

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

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

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

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

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

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

Fig. 3. BNF Grammar for Rule Cards.

3.3 Description of the SODOP Approach

The SODA approach is flexible and relatively easy to extend for SOA *patterns* instead of antipatterns. Indeed, the DSL and the underlying SOFA framework allow the integration of new metrics required for the specification of patterns. The approach proposed in this paper, called SODOP, introduces five new patterns, that we identified from the SOA literature. These patterns have been specified with rule cards by combining existing metrics along with eight newly defined ones—those are underlined in Figure 3 and are briefly described below. SODOP's three steps are described in Figure 4, and are similar to SODA's ones. The DSL grammar has been extended to allow more flexibility in the rule card specification. We add the possibility of combining two existing metrics with numeric operators to avoid the proliferation of new metrics and, thus, to provide ratios. The pattern rule cards specified in Step 1 are generated automatically into detection algorithms in Step 2, followed by the concrete detection of patterns on SOA systems in Step 3. The first specification step is thus manual, whereas the second and third are automated.

The following eight new metrics were defined. The *Execution Time (ET)* represents the time spent by a service to perform its tasks; it differs from the response time as it excludes the execution time of nested services. The *Number of Different Clients (NDC)* is the number of different consumers, thus multiple

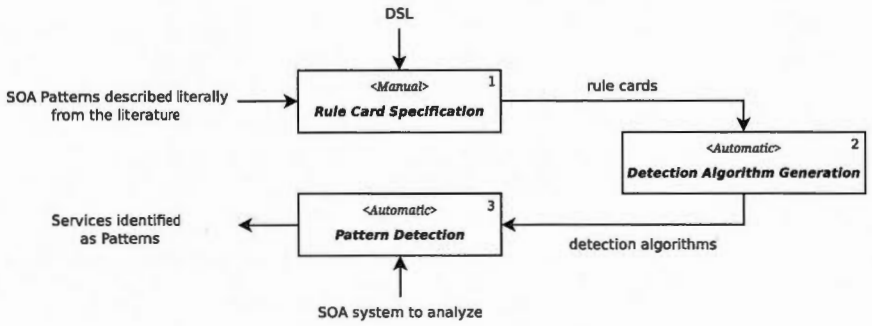

Fig. 4. The Three Steps of the SODOP Approach.

incoming calls from the same consumer are counted only once. By contrast, the *Number of Incoming Calls (NIC)* and *Number of Outgoing Calls (NOC)* refer to dynamic calls, thus possibly counting several times the same service. The *Delegation Ratio (DR)* represents the ratio of incoming calls that are relayed by a service. The *Service Reuse (SR)* is a dynamic metric that computes to what extent a service is reused; it is the ratio between the incoming calls (NIC) and the total number of calls in the system. The *Proportion of Outgoing Path Change (POPC)* computes the proportion of outgoing paths that change for a given incoming call. In other words, this proportion is zero if the incoming call and its underlying outgoing calls are always the same. Finally, the *Proportion of Signature Change (PSC)* computes the proportion of method signature change for a pair of incoming/outgoing calls. In other words, this metric represents the dissimilarity level between an incoming and outgoing method call; it is computed with the Jaro-Winkler similarity distance between method names [26].

3.4 Basic Service Pattern

When dealing with SOA pattern specification and detection, we want to specify the best fundamental characteristics every system designer or architect should take into account. Several principles, some of which are described in *SOA Patterns* [9], have to be considered for service design. Components reusability (SR) as well as high cohesion (COH) are common requirements in the design of general systems such as OO systems [5]. The dynamic nature of SOA systems introduces new non-functional requirements such as high availability (A) or low response time (RT). These metrics are combined in the rule card shown in Figure 9(a) for the specification of this *Basic Service* pattern.

3.5 Facade Pattern

A *Facade*, as illustrated in Figure 5, is used in SOA systems to get a higher abstraction level between the provider and the consumer layers. Fowler and Erl describe the pattern respectively as *Remote Facade* [10], *Decoupled Contract* or *Service Decomposition* [9] and give as example using it to wrap legacy systems. This pattern is similar to the *Facade* in OO systems because it hides implementation details [12] such as nested compositions and calls. It also provides

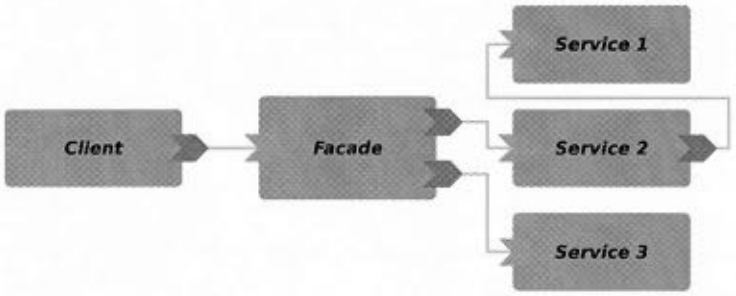

Fig. 5. Facade Pattern Example.

loosely coupled relationships with consumer services and let the implementation evolve independently, without breaking the client contract. Using this pattern, it is possible to decompose SOA systems following the principle of *separation of concerns*. It will thus be easier to reuse the different layers in other systems. A *Facade* can be responsible for orchestration, and can describe how composition of subsequent services can fulfill the client requirements. Given that the *Facade* acts as a front layer to several clients, we characterize its response time (RT) as high. Such a pattern is defined to hide implementation details from many services. Thus, its incoming outgoing calling ratio (NIC/NOC) is low because for one incoming call, the components tends to execute multiple outgoing calls. Finally, we assume that such a service has a high delegation ratio (DR) because it does not provide business logic directly but, instead, delegates to other services. Figure 9(b) shows the rule card specification for the *Facade* pattern.

3.6 Proxy Pattern

Fig. 6. Proxy Pattern Example.

The *Proxy* pattern, represented in Figure 6, is another well-known design pattern from OO systems, that adds an additional indirection level between the client and the invoked service. Its objective differs from a *Facade* because it can, for example, add new non-functional behaviors, which can cover security concerns such as confidentiality, integrity or logging execution calls for accountability goals. Different kinds of *Proxy* patterns exist, such as *Service Interceptor* [7] or *Service Perimeter Guard* [9], and they could all be specified with several distinct rule cards. Instead, we choose to specify a generic version of this pattern with the following characteristics. The proportion between incoming and outgoing calls (NIC/NOC) has to be equal to one because it acts only as a relay. Moreover, this relay property implies that incoming and outgoing method signatures have to be the same. The fact that the *Proxy* pattern generally adds

non-functional requirements to SOA systems also means that it can be involved in several scenarios. Thus, it has a high service reuse (SR) compared to other services. Figure 9(c) shows its underlying rule card.

3.7 Adapter Pattern

Fig. 7. Adapter Pattern Example.

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

3.8 Router Pattern

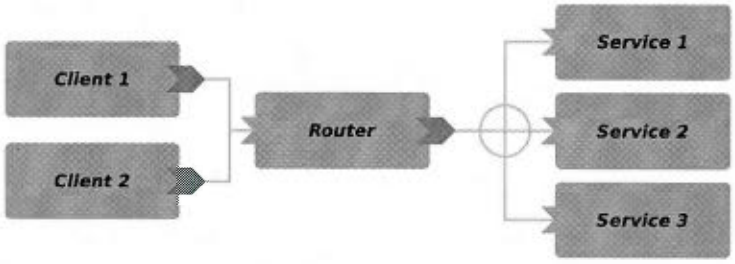

Fig. 8. Router Pattern Example.

The *Router* pattern, as illustrated in Figure 8, is similar to a network router that forwards packets according to different paths. A SOA *Router* distributes incoming calls to various destinations based on different criteria, which can be either the client identity or the call parameters. Some smart routers either detect paths on dynamic metrics such as availability or previous calls history and forward calls to the best matching service. The main criterion to consider is a change of outgoing paths for a specific incoming call, so a high proportion in path changes (POPC) can be significant. It may be interesting to see if some

specific clients use specific paths and then make the correlation with incoming parameters. Figure 9(e) shows the *Router* pattern rule card.

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

(a) Basic Service

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

(b) Facade

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

(c) Proxy

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

(d) Adapter

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

(e) Router

Fig. 9. Rule Cards for SOA Patterns.

4 Experiments

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

4.1 Assumptions

The experiments aim at validating the following three assumptions:

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

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

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

4.2 Analyzed Systems

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

4.3 Process

The process used for these experiments follows the three steps of the SODOP approach presented in Section 3. We first specified the rule cards representing the five SOA patterns described previously. Then, we generated automatically the detection algorithms in the second step. Finally, we applied them respectively on *Home-Automation* and *FraSCAti* to detect the SOA patterns specified. We validated the results by computing the precision—the proportion of true patterns in the detected patterns— and the recall—the proportion of detected patterns in all existing patterns. These validations were made through a manual and static analysis of each service in the systems under analysis. The computations were performed by two external software engineers to ensure the results were not biased. An additional feedback was given by the *FraSCAti* core team itself to strengthen the results.

4.4 Results

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

Details of the Results on *Home-Automation*

Four of the five specified SOA patterns were detected on *Home-Automation*— the *Adapter* pattern was not detected. The *patientDAO*, *communication* and *knxMock* components are detected as *Basic Service* pattern with a maximal cohesion (COH $\geq$ 0.34), high reuse values (SR $>$ 0.10) and very low response

time (RT < 0.25ms). According to the definition of the *Basic Service* pattern, these three components thus represent the services in the system that are the most well designed, as they appear to satisfy common software design principles. The *mediator* component is considered both a *Facade* and a *Router*. The *Facade* represents a service acting as a front layer to clients to hide a complex subsystem. Indeed, the delegation metric (DR = 1) of the *mediator* component always acts as a relay and tends to have six times more outgoing calls for each incoming one (NIC/NOC = 0.17), thus this traduces its high response time (RT = 2.8ms). The *mediator* has also been detected as a *Router* because of its high dynamic metric (POPC = 0.5). This value means that the *mediator* distributes to different outgoing paths half of its incoming calls. The *patientDAO* also matches the *Proxy* pattern because of its high reuse (SR = 0.24 compared to the median value of 0.06) and systematic incoming calls relay (NIC = NOC) with the same method signatures (PSC = 0). We can also observe that the time required for the detection of each pattern is on average 25ms, whereas the average execution time on a given set of scenarios is 6.73s. These values demonstrate the low impact of the pattern detection on the system execution, and thus on the results. Finally, the validation performed by the two experts lead to a 93.3% precision and 100% recall, which indicates that all existing patterns in *Home-Automation* have been detected, with high precision.

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

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

Details of the Results on *FraSCAti*

As shown in Table 2, the detection of patterns on *FraSCAti* returns more results than *Home-Automation*, i.e. more components are detected as patterns. This is partly explained by the size of *FraSCAti*, which is almost ten times larger than *Home-Automation*. Five components have been detected as matching the *Basic Service* pattern, because of their very high reusability (SR > 0.1 com-

pared to the median value of 0.003), high cohesion (COH > 0.48) and mostly very low response time (RT < 0.7ms). The core framework components, *FraSCAti*, *assembly-factory* and *composite-parser*, are detected as *Facade* as they are main entry points of the framework that relay every incoming calls (DR = 1). Their incoming outgoing call ratio (NIC/NOC = 0.46, 0.25 and 0.63) remains low compared to the median value of 1. They act as a *Facade* because they have among the highest response times (respectively 571ms, 181ms and 10ms) mainly due to their massive underlying calls. The *Proxy* pattern is involved in the three following components: *sca-interface*, *sca-implementation* and *component-factory*. The components have been identified as *Proxy* because they represent highly reused (SR > 0.5) relay services (NIC/NOC = 1) and they include the same method calls (PSC = 0). The only missing SOA pattern in *Home-Automation* and discovered in *FraSCAti* is the *Adapter*, seen in the *BindingFactory* component. It acts as an *Adapter* because it relays all its incoming calls (NIC/NOC = 1) and adapts the method calls to the underlying components (PSC = 1, which indicates a high proportion of signature change). Unlike in *Home-Automation*, no *Router* pattern has been detected. The average detection time required for our experiments is 97ms on average for a total time average of 10.9s. As for *Home-Automation*, the detection represents only 1% of the total system execution, and thus does not affect its performance, even with a relatively larger system. We reported those results to the *FraSCAti* core team and they confirmed all components detected as patterns. This leads to a precision of 100% for this detection. However, the recall value of 11.8% is low. Our detection algorithms thus failed at detecting all the existing components involved as patterns in the *FraSCAti* system.

Table 2. SOA Pattern Detection Results on the FraSCAti System.

PatternName	DetectedServices	Metrics			DetectTime	ExecTime	Precision	Recall
Basic Service	*sca-interface*	COH 0.48	RT 0.09ms	SR 0.1	241ms	11.34s	[5/5] 100%	n/a
	sca-interface-java	0.50	0.02ms	0.1				
	sca-impl.	0.48	0.67ms	0.0				
	sca-impl.-java	0.50	0.59ms	0.0				
	sca-comp.-service	0.48	0.25ms	0.48				
Facade	*FraSCAti*	NIC/NOC 0.46	DR 1.0	RT 571ms	57ms	10.62s	[3/3] 100%	[3/16] 18.7%
	assembly-factory	0.25	1.0	181ms				
	composite-parser	0.63	1.0	10ms				
Proxy	*sca-interface*	NIC/NOC 1.0	SR 0.11	PSC 0.0	65ms	10.72s	[3/3] 100%	[3/14] 21.4%
	sca-impl.	1.0	0.05	0.0				
	component-factory	1.0	0.11	0.0				
Adapter	*BindingFactory*	NIC/NOC 1.0	PSC 1.0		57ms	10.96s	[1/1] 100%	[1/14] 7.1%
Router	n/a	n/a			67ms	10.88s	[0/0] 100%	[0/7] 0.0%
Average					97ms	10.90s	100%	11.8%

4.5 Discussion

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

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

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

A3. Performance: *The time needed by the detection algorithms must not impact the performance of the analyzed system.* As shown in Tables 1 and 2, no matter which SOA patterns or how many metrics are computed, the detection time remains low compared to the execution time and thus does not impact the system under analysis. As a first analysis, we find the only affecting property is the number of services involved in the SOA system under analysis, because all the metrics are computed against each of them. *FraSCAti* has around eight times more components than *Home-Automation*, which explains the proportional time needed to run the metrics computation (around 1% of the execution time). Because the experiments are run locally, the execution time is also highly dependent on the computer computational power. In these experiments, an Intel E5345 CPU with 4GB of RAM was used.

4.6 Threats to Validity

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

described in the literature. Moreover, although we only defined five SOA patterns in the form of rule cards, they are representative according to the literature. Indeed, even if SOA catalogs mainly define patterns for specific technologies, we tried to specify meaningful technology-agnostic SOA patterns.

5 Conclusion and Future Work

SOA patterns are proven good practices to solve known and common problems when designing software systems. Indeed, our three steps SODOP approach consists in the specification and detection of SOA patterns to assess the design and QoS of SOA systems. The first step consists in specifying rule cards—set of rules, combining static and dynamic metrics—for each pattern. Five patterns were described in our study, involving 22 different static and dynamic metrics, including eight newly defined dynamic metrics. The second step consists in generating automatically detection algorithms from rule cards, and applying them on SOA systems in the third step. We validated our approach using two SCA systems, *Home-Automation*—a system that provides 13 services for domotic tasks—and *FraSCAti*—a SCA standard implementation that provides 91 components. The experiments showed that we can obtain high precision and recall values under the condition that execution scenarios are exhaustive.

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

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

References

1. Acceleo code generator tool, `http://www.acceleo.org/`
2. Eclipse modeling framework project, `http://www.eclipse.org/modeling/emf/`
3. Antoniol, G., Fiutem, R., Cristoforetti, L.: Design Pattern Recovery in Object-Oriented Software. In: 14th IEEE Intl. Conf. on Prog. Comprehension. pp. 153–160 (Jun 1998)
4. Banker, R.D., Datar, S.M., Kemerer, C.F., Zweig, D.: Software complexity and maintenance costs. Comm. of the ACM 36(11), 81–94 (Nov 1993)
5. Basili, V., Briand, L., Melo, W.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 22(10), 751–761 (1996)
6. Chappell, D.: Introducing SCA (2007), `http://www.davidchappell.com/articles/introducing_sca.pdf`
7. Daigneau, R.: Service Design Patterns. Addison-Wesley (2011)
8. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Improving Behavioral Design Pattern Detection through Model Checking. In: 14th European Conf. on Soft. Maintenance and Reengineering. pp. 176–185. IEEE Comp. Soc. (Mar 2010)

9. Erl, T.: SOA Design Patterns. Prentice Hall PTR (2009)
10. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
11. Galster, M., Avgeriou, P.: Qualitative Analysis of the Impact of SOA Patterns on Quality Attributes. In: 12th Intl. Conf. on Quality Software. pp. 167–170. IEEE (Aug 2012)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
13. Guéhéneuc, Y.G., Antoniol, G.: DeMIMA: A Multilayered Approach for Design Pattern Identification. IEEE Trans. on Soft. Eng. 34(5), 667–684 (Sep 2008)
14. Hansen, M.D.: SOA Using Java Web Services. Prentice Hall (2007)
15. Heuzeroth, D., Holl, T., Hogstrom, G., Löwe, W.: Automatic design pattern detection. In: Intl. Symp. on Micromechatronics and Human Science. pp. 94–103. IEEE Comp. Soc. (2003)
16. Hohpe, G., Easy, C.: SOA Patterns–New Insights or Recycled Knowledge. Tech. rep. (2007)
17. Hu, L., Sartipi, K.: Dynamic Analysis and Design Pattern Detection in Java Programs. In: 20th Intl. Conf. on Soft. Eng. and Data Eng. pp. 842–846 (2008)
18. Ka-Yee Ng, J., Guéhéneuc, Y.G., Antoniol, G.: Identification of Behavioral and Creational Design Patterns through Dynamic Analysis. In: 3rd Intl. Work. on Progr. Comprehension through Dynamic Analysis. pp. 34–42. John Wiley (2007)
19. Milanovic, N.: Service Engineering Design Patterns. Second IEEE Intl. Symp. on Service-Oriented System Eng. pp. 19–26 (Oct 2006)
20. Moha, N., Nayrolles, M., Joyen-Conseil, B., Guéhéneuc, Y.G., Baudry, B., Jézéquel, J.M.: Specification and Detection of SOA Antipatterns. In: Tenth Intl. Conf. on Service Oriented Computing. pp. 1–16 (2012)
21. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Proc. Conf. on Soft. Maint. pp. 337–344. IEEE Comp. Soc. Press (1992)
22. Rotem-Gal-Oz, A.: SOA Patterns. Manning Publications (2012)
23. Schulte, R.W., Natis, Y.V.: Service Oriented Architectures, Part 1. Tech. rep., Gartner (1996)
24. Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable SCA Applications with the FraSCAti Platform. In: 2009 IEEE Intl. Conf. on Services Computing. pp. 268–275. IEEE Computer Society (Sep 2009)
25. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.: Design Pattern Detection Using Similarity Scoring. IEEE Trans. on Soft. Eng. 32(11), 896–909 (Nov 2006)
26. Winkler, W.E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. (Nov 1989)
27. Yousefi, A., Sartipi, K.: Identifying distributed features in SOA by mining dynamic call trees. In: 27th IEEE Intl. Conf. on Soft. Maint. pp. 73–82. IEEE (Sep 2011)

BIBLIOGRAPHIE

Antoniol, G., R. Fiutem et L. Cristoforetti. 1998. « Design Pattern Recovery in Object-Oriented Software ». In *14th IEEE Intl. Conf. on Prog. Comprehension*, p. 153–160.

Banker, R. D., S. M. Datar, C. F. Kemerer et D. Zweig. 1993. « Software complexity and maintenance costs ». *Comm. of the ACM*, vol. 36, no. 11, p. 81–94.

Basili, V., L. Briand et W. Melo. 1996. « A validation of object-oriented design metrics as quality indicators ». *IEEE Transactions on Software Engineering*, vol. 22, no. 10, p. 751–761.

Chappell, D. 2007. « Introducing SCA », <`http://www.davidchappell.com/articles/introducing_sca.pdf`>.

Daigneau, R. 2011. *Service Design Patterns*. Addison-Wesley.

De Lucia, A., V. Deufemia, C. Gravino et M. Risi. 2010. « Improving Behavioral Design Pattern Detection through Model Checking ». In *14th European Conf. on Soft. Maintenance and Reengineering*, p. 176–185. IEEE Comp. Soc.

Demange, A., N. Moha et G. Tremblay. 2013. « Detection of SOA Patterns ». In *11th International Conference on Service Oriented Computing*, sous la dir. de S. Basu, C. Pautasso, L. Zhang et X. Fu. T. 8274, série *Lecture Notes in Computer Science*, p. 114—130, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dudney, B., S. Asbury, J. Krozak et K. Wittkopf. 2003. *J2EE AntiPatterns*. John Wiley & Sons Inc.

Erl, T. 2009. *SOA Design Patterns*. Prentice Hall PTR.

Fowler, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

Gamma, E., R. Helm, R. Johnson et J. Vlissides. 1994. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Guéhéneuc, Y.-G., et G. Antoniol. 2008. « DeMIMA : A Multilayered Approach for Design Pattern Identification ». *IEEE Trans. on Soft. Eng.*, vol. 34, no. 5, p. 667–684.

Hansen, M. D. 2007. *SOA Using Java Web Services*. Prentice Hall.

Heuzeroth, D., T. Holl, G. Hogstrom et W. Löwe. 2003. « Automatic design pattern detection ». In *Intl. Symp. on Micromechatronics and Human Science*, p. 94–103. IEEE Comp. Soc.

Hohpe, G., et C. Easy. 2007. SOA Patterns–New Insights or Recycled Knowledge. Rapport.

Hu, L., et K. Sartipi. 2008. « Dynamic Analysis and Design Pattern Detection in Java Programs ». In *20th Intl. Conf. on Soft. Eng. and Data Eng.*, p. 842–846.

« Frascati open-source SCA implementation ». 2009. <http://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti>.

Ka-Yee Ng, J., Y.-G. Guéhéneuc et G. Antoniol. 2007. « Identification of Behavioral and Creational Design Patterns through Dynamic Analysis ». In *3rd Intl. Work. on Progr. Comprehension through Dynamic Analysis*, p. 34–42. John Wiley.

Koenig, A. 1995. « Patterns and Antipatterns ». *Journal of Object-Oriented Programming 8*, p. 46–48.

Milanovic, N. 2006. « Service Engineering Design Patterns ». *Second IEEE Intl. Symp. on Service-Oriented System Eng.*, p. 19–26.

Moha, N., M. Nayrolles, B. Joyen-Conseil, Y.-G. Guéhéneuc, B. Baudry et J.-M. Jézéquel. 2012. « Specification and Detection of SOA Antipatterns ». In *Tenth Intl. Conf. on Service Oriented Computing*, p. 1–16.

Oman, P., et J. Hagemeister. 1992. « Metrics for assessing a software system's maintainability ». In *Proc. Conf. on Soft. Maint.*, p. 337–344. IEEE Comp. Soc. Press.

« Eclipse modeling framework text plugin ». 2013. <http://www.emftext.org/>.

« Eclipse modeling framework project ». 2011. <http://www.eclipse.org/modeling/emf/>.

Rasool, G., I. Philippow et P. Mäder. 2010. « Design pattern recovery based on annotations ». *Advances in Engineering Software*, vol. 41, no. 4, p. 519–526.

Rotem-Gal-Oz, A. 2012. *SOA Patterns*. Manning Publications.

Schulte, R. W., et Y. V. Natis. 1996. Service Oriented Architectures, Part 1. Rapport, Gartner.

Seinturier, L., P. Merle, D. Fournier, N. Dolet, V. Schiavoni et J.-B. Stefani. 2009. « Reconfigurable SCA Applications with the FraSCAti Platform ». In *2009 IEEE Intl. Conf. on Services Computing*, p. 268–275. IEEE Computer Society.

« Acceleo code generator tool ». 2010. <http://www.acceleo.org/>.

Tsantalis, N., A. Chatzigeorgiou, G. Stephanides et S. Halkidis. 2006. « Design Pattern Detection Using Similarity Scoring ». *IEEE Trans. on Soft. Eng.*, vol. 32, no. 11, p. 896–909.

Wendehals, L., et A. Orso. 2006. « Recognizing behavioral patterns at runtime using finite automata ». In *2006 International Workshop on Dynamic systems analysis*, p. 33–40, New York, New York, USA. ACM Press.

Winkler, W. E. 1990. « String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage ». In *Section on Survey Research Methods*, p. 354–359.

Yousefi, A., et K. Sartipi. 2011. « Identifying distributed features in SOA by mining dynamic call trees ». In *27th IEEE Intl. Conf. on Soft. Maint.*, p. 73–82. IEEE.