Validation of reasoning engines an adaptation mechanisms for self-adaptive systems

Freddy Munoz

N° d'ordre : 4186 ANNÉE 2010

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Freddy Munoz

préparée à l'unité de recherche UMR 6074 IRISA
Institut de Recherche en Informatique et Systmes Alátoires
Composante Universitaire : IFSIC

Validation of reasoning engines and adaptation mechanisms for *self*-adaptive systems

Thèse soutenue à Rennes le 29 Septembre 2010

devant le jury composé de :

Olivier RIDOUX
Professeur à l'Université de Rennes 1 / président

Yves LE TRAON
Professeur à l'Université de Luxembourg / examinateur

Robert FRANCE
Professeur, Colorado State University / rapporteur

Franck BARBIER
Professeur à l'Université de Pau et des Pays de l'Adour / rapporteur

Jean-Marc JÉZÉQUEL
Professeur à l'Université de Rennes 1 / directeur de thèse

Benoit BAUDRY
Chargé de Recherche, INRIA Rennes Bretagne Atlantique/ Encadrant de thèse

Con fe lo imposible soñar
al mal combatir sin temor
triunfar sobre el miedo invencible
en pie soportar el dolor

Amar la pureza sin par
buscar la verdad del error
vivir con los brazos abiertos
creer en un mundo mejor

Es mi ideal
la estrella alcanzar
no importa cuan lejos
se pueda encontrar
luchar por el bien
sin dudar ni temer
y dispuesto al infierno llegar si lo dicta el deber

Y yo sé
que si logro ser fiel
a mi sueño ideal
estará mi alma en paz al llegar
de mi vida el final

Será este mundo mejor
si hubo quien despreciando el dolor
combatió hasta el último aliento

Con fé lo imposible soñar
y la estrella alcanzar

El sueño imposible – *Don Quijote de la Mancha*

Remerciements

First, I would like to thank the members of my examination jury. My examiners Robert France, Frank Barbier, and Yves Le-Traon, who where generous with their comments and feedback on my research. I value your assessment and input to its improvement. Olivier Rideaux for agreeing to preside over this thesis jury. I was flattered by the quality of the jury, and grateful for the insight of their feedback and questions. Jean-Marc Jézéquel, who accepted to grant me the opportunity to work with Triskell and who directed this thesis. I appreciate your trust on me and your leadership capabilities.

I must specially thank Benoit Baudry, who four years before accepted to advise. Thank you for being my friend and letting me the free to drive my own ideas while advising me where was better to go. Your thoughtful advise and friendship was and is a real guide to me.

Je remercie les membres de l'équipe Triskell pour leur chaleureuse accueil et support pendants mes quatre années avec eux. Je voudrais bien remercier mes collègues de bureau Cyril Faucher, Romain De-Lamaire, Gregory Nain et Juan Jose Cadavid ; c'était un plaisir partager un espace de travaille avec vous. Je voudrai aussi remercier Sagar Sen et Brice Morin qui ont toujours étés disponibles pour des discussions enrichissantes sur nos études et sur l'avenir de nos thèses. Je remercie aussi Loïc Lesage pour m'a aider quand j'avais besoin d'une main avec les affaires administratives.

Finalmente debo agradecer a aquellos que me han apoyado durante todo estos años, se han alegrado con mis logros y no han dejado de creer en mis ideas. Mi esposa Constanza, quien me dio apoyo incondicional en esta aventura. Mis padres – Freddy y Cynthia – y mis hermanas quienes me siempre me han brindado su apoyo y confianza para cumplir mis sueños. Mi tíos quienes me brindaron el apoyo fundamental para comenzar la empresa que culmina con esta tesis. Mis abuelos quienes como mis padres han creído en mis capacidades y me han apoyado a seguir adelante. Finalmente, debo agradecer a mis amigos –Leo, Daniel y Juan Pablo–, quienes me apoyaron y entendieron los días de larga ocupación, aceptaron tener largas fatigantes reuniones y nunca criticaron mis ausencias.

This thesis no only belong to me, but to all those who supported and encouraged me to get here. *Thank you all*.

Per Ardua Ad Astra – Freddy Munoz

Contents

Sistèmes *auto*-adaptatives — sommaire en français

0.1 Introduction generale

Les systèmes logiciels ont propulsé la course des humains pour l'automatisation en nous aidant à automatiser des tâches répétitives pourtant complexes et améliorer la qualité de vie. De nos jours, nous voyons les systèmes logiciels qui aident des ingénieurs à commander des centrales nucléaires, des pilotes à diriger des avions de 500 passagers, des médecins à chercher les informations de patients en un instant, des hommes d'affaires à fermer des contrats de plusieurs millions de dollars, et ainsi de suite. Ces systèmes jouent un rôle essentiel dans l'infrastructure des sociétés. Pourtant, on attend aujourd'hui encore plus de ces systèmes.

Dans le futur, on attend des systèmes logiciels qu'ils s'adaptent et répondent mieux aux besoins du monde dynamique dans lequel nous vivons [134], et fournir des bénéfices au delà des limites actuelles. De tels systèmes constituent la prochaine étape dans l'automatisation et devront offrir flexibilité, fiabilité et robustesse, parmi d'autres propriétés importantes. Ces derniers systèmes sont appelés auto adaptatifs parce qu'ils peuvent changer leur structure et comportement pour s'adapter à un monde et à ses besoins en changement continu. Ces systèmes reproduisent, aussi bien que possible, les capacités des systèmes biologiques à s'adapter aux environnements en cours d'évolution. Les domaines d'application potentiels des systèmes auto adaptatifs sont sans fin et incluent entre autres, la gestion des crises [114, 112], l'exploration de l'espace [72], la domotique [157, 170, 160], le contrôle de transport, les applications d'affaires [27] et d'amusement, etc.

Un système auto adaptatif est un système logiciel capable de : (i) *percevoir l'environnement et les changements environnementaux.* Par exemple, dans la domotique, ces systèmes peuvent percevoir des éléments environnementaux tels que la température, l'humidité, l'éclairage, etc. (ii) *raisonner sur les changements environnementaux et de décider comment s'auto modifier pour s'adapter aux changements.* Par exemple, quand la température descend en dessous de 10 degrés Celsius, le système décide d'allumer le chauffage. (iii) *re-configurer sa structure interne en accord avec les décisions prises pour s'adapter.* Par exemple, suite a la

décision d'activer le chauffage, le système connecte le module de contrôle des radiateurs et les allume.

Construire et valider des systèmes *auto*-adaptatifs propose des défis de conception, modélisation, et implantation pour chercheurs et ingénieurs [186, 124, 44]. Ces défis sont liés à la complexité de ces systèmes dans plusieurs dimensions. La complexité de prendre des décisions sur un grand nombre de conditions environnementales. La complexité de modifier le système pendant son exécution et assurer qu'il continuera à fonctionner après les modifications. La complexité de gérer un large nombre de configurations, et ainsi de suite. D'autres défis sont liés à la validité et à la conformité de ces systèmes : assurer que ces systèmes perçoivent correctement l'environnement et les modèles qui le représentent ; assurer que les décisions prises par ces systèmes sont toujours correctes et que l'implantation du système est fiable et sûre ; assurer que ces systèmes ne cesseront pas de fonctionner. Ce dernier défi est critique.

Des fautes dans des systèmes *auto*-adaptatifs peuvent amener des conséquences légères, comme laisser l'individu λ dans l'obscurité, ou graves, comme surchauffer la maison et mettre en péril la vie de ses occupants. Notre société peut bénéficier énormément de ces systèmes auto adaptatifs, cependant ça n'arrivera que si l'on peut assurer qu'ils se comporteront de manière sûre, que (i) *ces décisions seront comme prévues*, et que (ii) *ces adaptations ne produiront pas d'effet de bord non désirés ou ne détruiront pas le système*.

Tel est le problème qu'on soulève dans cette thèse : **l'assurance du raisonnement et l'adaptation des systèmes *auto*-adaptatifs**. Premièrement, on adresse l'assurance du processus de décision en proposant une technique innovatrice pour l'échantillonnage de l'environnement. Deuxièmement, on adresse l'assurance des mécanismes d'adaptation par aspects en proposant un framework de spécification pour les interactions des aspects. Cette thèse contribue à la connaissance produite par des nombreux projets européens [187, 197, 82], ainsi que plusieurs groupes de recherches autour du monde qui ont adressé les différents défis introduits par les systèmes adaptatifs.

0.2 Sistèmes *auto*-adaptatives

Les systèmes auto adaptatifs répondent au besoin de contrôler la complexité et réagir face aux changements des environnements opérationnels [43]. Ces systèmes sont censés s'exécuter pendant des périodes très longues (parfois même éternellement), répondre aux environnements en cours d'évolution et continuer à fonctionner et à fournir le meilleur service possible. Ainsi, ils doivent changer leur structure interne et leurs propriétés afin de continuer à fonctionner dans le temps.

En général, un système auto adaptatif se compose de quatre éléments, la Figure 1 les illustre :

(1) Un ensemble de *sondes*, qui détectent et traduisent les fluctuations environnementales dans une représentation de l'environnement. Les chercheurs ont proposé une

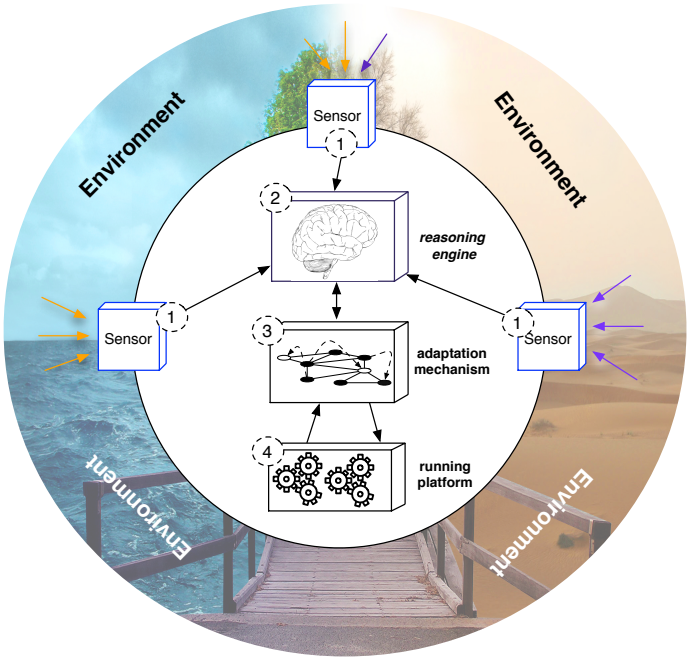

FIGURE 1 – *Self*-adaptive system

série de représentations [106, 230, 42, 81, 18] d'environnement qui peuvent changer selon la nature de l'environnement physique (le cas échéant).

(2) Un *moteur de raisonnement*, qui commande le rapport entre les variations environnementales et les changements internes du système. Ce moteur de raisonnement prend des décisions basées sur une série de conditions environnementales que les sondes lui envoient. Ces décisions décrivent les actions atomiques qui constituent les modifications globales dans la structure ou le comportement du système. Il existe principalement trois techniques de raisonnement : (i) *le raisonnement à base de règles* [91], qui emploie une série de règles pour dériver une décision, (ii) *le raisonnement à base de buts* [105], qui emploie des buts de haut niveau pour dériver des décisions, et (iii) *le raisonnement à base de fonction de coût* [125], qui emploie une fonction de coût pour mesurer, classer, et choisir une décision parmi d'autres. D'autres techniques de raisonnement ont été proposées récemment, techniques qui se servent des modèles stochastiques [35, 75, 75] ou mécanismes d'essai / erreur [107] pour conduire la prise des décisions.

(3) Une *plateforme d'exécution* qui correspond à la structure exécutable du système, qui fournit des fonctionnalités. Il existe une série de plateformes d'exécution, des programmes codés en dur [28] jusqu'au système à base de composants [31, 60, 177, 33].

(4) Un *mécanisme de re-configuration* ou *adaptation* qui exécute les décisions prises par le moteur de raisonnement et qui modifie la structure et le comportement de la plateforme d'exécution. Une large variété de mécanismes d'adaptation et plateformes d'exécutions ont été proposés, des adaptations codées en dur [97], des adaptations réflexives [7], des adaptations par aspects [220], jusqu'à des mécanismes plus abstraits, comme des mécanismes d'adaptation dirigés par les modèles [156] capables de gérer la plateforme d'exécution et ses changements en utilisant des modèles.

Dans cette thèse, on s'intéresse à la validation de deux de ces composants, les moteurs de raisonnement et les mécanismes d'adaptation. Plus précisément, on s'intéresse aux moteurs de raisonnement en général, comme pilotes de la prise de décisions, et aux mécanismes d'adaptation dirigés par aspects.

La programmation orientée aspects [126] (AOP) est une technique de programmation que permet aux développeurs d'utiliser des parties de code appelées aspects pour modifier à la volée les modules du système. AOP se sert des points de coupure, ou pointeurs, qui désignent des points bien définis dans le système, et des advices, qui implantent la modification du comportement. Comme mécanisme d'adaptation, AOP permet aux développeurs et ingénieurs d'implanter des modifications structurales et comportementales dans des aspects, pour après, si besoin, tisser ou défiler les aspects dynamiquement au temps d'exécution pour adapter le système.

0.3 Défis por la validation des systèmes *auto*-adaptatives

La validation des systèmes auto adaptatifs est un processus qui assure que ces systèmes s'adaptent correctement à n'importe quel changement environnemental auquel il puisse faire face. Cette thèse est concernée par la validation des moteurs de raisonnement et des mécanismes d'adaptation par l'AOP.

La validation des moteurs de raisonnement consiste à vérifier qu'ils sont capables de prendre la bonne décision face à n'importe quel changement environnemental, c'est-à-dire exécuter le moteur de raisonnement contre tout environnement possible et ses variations, et observer s'il décide correctement. S'il prend les bonnes décisions, alors le moteur de raisonnement est valide, autrement il est défectueux.

La validation des mécanismes d'adaptation pas l'AOP consiste à vérifier si le tissage des aspects dans le système de base produit les adaptations correctes, c'est-à-dire vérifier : (1) que les nouveaux aspects tissés dans le système ne dégradent pas des fonctionnalités existantes, et (2) que les fonctionnalités ou les modifications introduites par les aspects s'exécutent comme prévu.

La validation de chacun de ces éléments impose plusieurs défis que l'on récapitule dans le reste de cette section.

0.3.1 Défis de la validation des moteurs de raisonnement

Les moteurs de raisonnement prennent des décisions basées sur des conditions environnementales passées et courantes. L'environnement lui-même est représenté par un ensemble de variables de raisonnement, dont leur valeurs décrivent les conditions possibles (ou des instances) que l'environnement peut adopter. Ces variables de raisonnement créent un espace qui contient toutes les conditions environnementales possibles, connu sous le nom d'espace de raisonnement. Cet espace incarne les différentes interactions entre les valeurs des différentes variables de raisonnement, que l'on référence en tant qu'interactions inter-variables. L'espace de raisonnement comporte aussi les différentes variations environnementales qui peuvent se produire au cours du temps (temporalité). Ces variables produisent des interactions entre les valeurs à l'intérieur de chaque variable de raisonnement, que l'on référence en tant qu'interactions intra-variables.

Les interactions intra et inter variables forment l'espace de raisonnement et le rendent très grand. Par exemple, considérez un environnement modelé par 10 variables d'environnement avec 3 valeurs chacune. Le nombre total de conditions environnementales pour cet environnement est $3^{10} = 59.049$ intenses. Autrement dit, cet espace de raisonnement incarne 59.049 interactions inter-variables. En fait, si nous considérons la dimension temporelle de l'environnement, l'espace de raisonnement est encore plus grand. Considérez que cet espace de raisonnement contemple seulement deux intenses sur son histoire (ou l'interaction inter-variable entre deux valeurs de la même variable). Alors, la taille de l'espace de raisonnement est $59.049^2 = 3.5 \times 10^9$, c'est-à-dire que cet espace de raisonnement comporte 3.5×10^9 interactions intra-variables.

Valider que le moteur de raisonnement prenne les décisions correctes implique de vérifier que ses décisions sont correctes pour l'espace entier de raisonnement. Or, ce n'est pas toujours possible. Par exemple, considérez l'espace de raisonnement précédent, la validation d'un moteur de raisonnement sur cet espace implique de vérifier s'il prend les décisions correctes sur chacune des $3.5\ time 10^9$ variations environnementales, soit un processus qui peut s'exécuter au cours de plusieurs siècles avant de finir. Par conséquent, la validation de l'espace de raisonnement au-dessus de chaque état environnemental possible est en général impraticable.

Les techniques d'état de l'art existantes proposent de ramener le nombre de conditions environnementales au contrôle. Une technique appelée mixed level covering arrays (MCA) [55] couvre effectivement les interactions qui se produisent entre les différentes variables (inter-variables) de raisonnement. MCA réduit nettement le nombre de conditions environnementales nécessaires pour valider le moteur de raisonnement, cependant, il ne couvre pas la dimension temporelle du raisonnement et ne traite pas les interactions intra-variables.

Un défi important pour valider des moteurs de raisonnement est de pouvoir gérer les différentes conditions environnementales (interactions inter-variables) et la dimension temporelle des variations sur ces conditions (les interactions intra-variables).

0.3.2 Défis de la validation de l'AOP

L'AOP a une série de problèmes qui empêchent son adoption comme mécanisme d'adaptation et posent des obstacles à la validation. Le premier problème, connu sous le nom de *paradoxe d'évolution* [217, 132, 150], se produit quand les aspects et le système de base évoluent séparément et provoquent le tissage des aspects dans des modules non souhaités, ou que des aspects ne soient pas tissés dans les modules souhaités. Ce problème provient de l'insuffisance du langage de point de coupe actuel pour abstraire des propriétés structurales du système de base. Le deuxième problème, connu sous le nom d'*interférence d'aspect* [121], se produit quand plusieurs aspects sont tissés dans le même point de système, ou quand un aspect décommande l'effet d'autres aspects. Le troisième problème se fonde sur deux propriétés d'AOP : l'*inconscience* [80] (la capacité des aspects d'exécuter du comportement sans être demandé) et l'*envahissement* [164] (ou la capacité des aspects à casser l'encapsulation orientée objet). L'envahissement est une propriété puissante qui permet que l'AOP soit utilisée comme mécanisme d'adaptation puisqu'il laisse remplacer et augmenter le comportement du système. Cependant, une fois utilisé inconsciemment, il peut également introduire du comportement et des effets de bord non désirés sans que les développeurs s'en rendent compte.

Ces problèmes impactent négativement la validation de l'AOP, les effets secondaires peu désirés, l'interférence entre les aspects, et l'envahissement non contrôlé rendent difficile le fait de s'assurer que les fonctionnalités sont préservées quand de nouveaux aspects sont tissés avec le système de base. Par conséquent, pour vérifier si les nouveaux aspects peuvent préserver les fonctionnalités du système, il est nécessaire d'exécuter de nouveau les test fonctionnels. Si le système montre des déviations par rapport au comportement correct, les problèmes de l'AOP ne permettent pas facilement de savoir si les aspects introduisent des fautes.

Les techniques d'état de l'art ont abordé ces problèmes en caractérisant le comportement d'aspect [196, 122, 49, 77], proposant des directives [150, 217, 127] et des systèmes modulaires [5, 98, 143, 139] pour des programmes orientés aspect. La caractérisation d'aspect fournit un moyen de spécifier le comportement des aspects en ce qui concerne leur interférence et l'influence des aspects sur le programme de base. Cependant, la plupart des caractérisations fournissent seulement l'appui pour l'analyse théorique [196, 122, 77] ou caractérisent les aspects à gros grain [49]. Ceci ne garantit pas que l'aspect ne sera pas tissé ou qu'ils n'auront aucun effet secondaire. Les directives sont des ensembles de principes pour guider les développeurs face aux aspects, elles proposent des règles à suivre pour rendre les développeurs conscients des aspects et de ces effets dans leur code. Cependant, ils ne posent pas d'attachement aux développeurs, et ne proposent pas de garanties sur le comportement des développeurs et les aspects qu'ils écrivent. Les systèmes modulaires permettent aux développeurs et ingénieurs de spécifier finement les points du système qui sont ouverts pour tisser des aspects. Bien que les systèmes modulaires spécifient les points ouverts dans le système, ils permettent aussi que n'importe quel aspect puisse être

tissé sur ces points ouverts. Les systèmes modulaires adressent proprement les problèmes d'envahissement et d'inconscience mais ignore la possibilité que différents aspects puissent interférer avec le système de base.

Des défis importants pour la validation du mécanisme d'adaptation par AOP sont : (i) *le support pour l'évolution des programmes orientés par aspect*, (ii) *le contrôle de l'envahissement des aspects*, et (iii) *la gestion des interactions d'aspect*. Relever ces défis fournira les moyens aux développeurs et ingénieurs de savoir en avance si les aspects ont introduit un certain danger.

0.4 Contributions de cette these

Cette thèse comporte deux contributions majeures à la validation des systèmes auto-adaptatifs, chacune de ces contributions relevant les défis précédemment énumérés. La première contribution est une technique de sélections de données pour valider des moteurs de raisonnement. La deuxième contribution est un framework de spécifications pour les mécanismes d'adaptation par AOP. Dans le reste de cette section je récapitule chacune de ces contributions.

0.4.1 Test des moteurs de raisonnement

La validation des moteurs de raisonnement pose un défi en raison du grand nombre de conditions environnementales possibles à vérifier. Les techniques existantes ne gèrent pas la dimension temporelle de l'espace de raisonnement. Dans cette thèse on introduit le multi-dimensional covering array (MDCA), une technique qui peut effectivement gérer la dimension temporelle de l'espace de raisonnement (interactions intra-variables) et aussi gérer les interactions entre les variables. MDCA étend une technique existante, MCA, et préserve ses avantages. L'idée derrière MDCA est de couvrir en même temps les interactions qui se produisent entre les variables (interactions inter-variables) et les interactions entre les valeurs de chaque variable de raisonnement (interactions intra-variables) qui peuvent influencer la prise de décision.

MDCA échantillonne une quantité limitée de conditions environnementales pour valider des moteurs de raisonnement. L'hypothèse que soutient MDCA est que les conditions échantillonnées caractérisent les interactions de l'espace de raisonnement de telle manière que si la prise de décisions contient des défauts ou des déviations par rapport à la décision prévue, elles seront indiquées.

Essentiellement, un MDCA est un tableau de dimensions $N \times k$ qui satisfait les propriétés suivantes : (i) *les valeurs dans chaque file correspondent aux valeurs d'une variable précise* ; (ii) *pour chaque file, les* u*-colonnes consécutives, comportent toutes les* u*-tuples des valeurs de la variable correspondant pour la file* ; (iii) *les u-colonnes consécutives de chaque*

$t \times N$ *sous tableau comportent toutes les* t*-tuples de* u*-tuples des valeurs pour les variables correspondant aux* t *files.*

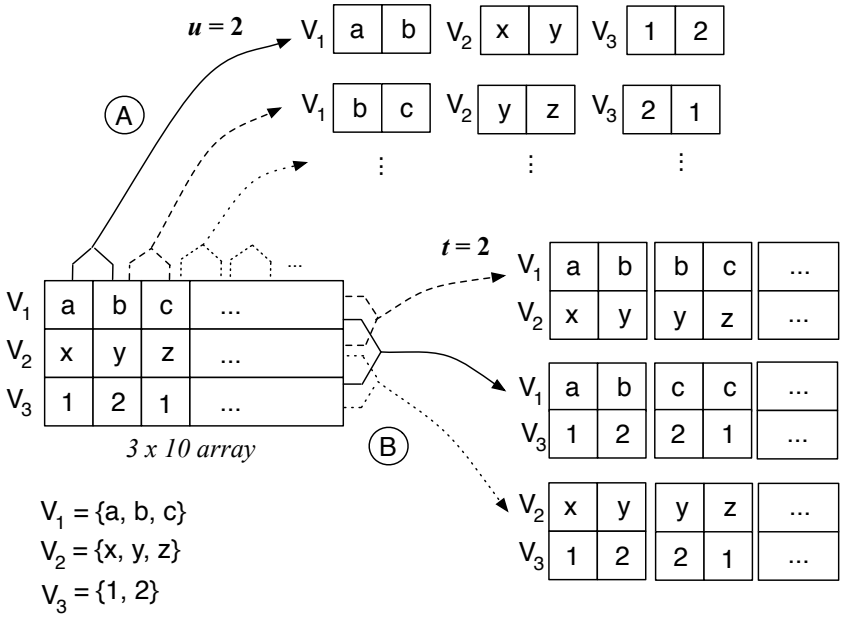

FIGURE 2 – Graphical view of an MDCA for 3 variables.

Notez que chaque file du MDCA contient une série des valeurs pour une variable de raisonnement, tandis que chaque colonne contient une valeur pour cette variable précise. Par conséquent, chaque colonne représente une instance d'environnement, l'ordre d'apparition de chaque instance est important puisque cet ordre représente les différentes transitions entre variables.

La Figure présente graphiquement un MDCA pour 3 variables. Les valeurs de t et de u s'appellent force, et force d'enchaînement, et représentent le nombre de variables que le MDCA considère pour les interactions inter-variables et le nombre de valeurs pour les interactions intra-variables. Dans la figure 2, le tableau a les valeurs $t = 2$ et $u = 2$, cela signifie que le MDCA considère les interactions entre les valeurs de deux variables différentes (inter), et les interactions entre les valeurs de deux valeurs différentes de la même variable (intra). La propriété (i) assure que chaque file contient les valeurs d'une variable de raisonnement seulement. La propriété (ii) assure que toutes les interactions intra-variables entre les valeurs de chaque variable de raisonnement sont présentes dans le tableau, la figure 2 (a) montre les interactions intra-variables entre 2 valeurs de chaque variable. La propriété (iii) assure que toutes les interactions inter-variables entre les variables sont présentes dans le tableau. La figure 2 (b) montre les interactions inter-variables entre les paires de diffé-

rentes variables. Notez que le défi est de construire des MDCAs avec le plus petit nombre de colonnes possible.

MDCA décrit les propriétés qu'un tableau doit satisfaire pour assurer la couverture des interactions intra et inter variable. Afin de démontrer la faisabilité de MDCA, on a codé sa construction comme problème d'optimisation fonctionnel qui cherche à optimiser la satisfaction des propriétés qui définissent un MDCA. Pour résoudre ce problème on emploie trois méta-heuristiques – algorithme génétique [96] (GA), algorithme bactériologique [12] (BA), et un algorithme hybride (HA) – qui utilisent différentes stratégies pour construire des tableaux satisfaisant les propriétés du MDCA. Le GA est une technique qui simule l'évolution d'une population d'individus qui évolue vers une solution optimale par mutations et sélections naturelles génétiques. Le BA est une adaptation de GA pour simuler l'évolution d'un groupe de bactéries. BA sélectionne les batteries qui peuvent améliorer la solution globale, pendant que les autres qui ne l'améliorent pas sont mutés puis jetés. L'HA est une combinaison de GA et BA qui construit une solution avec une stratégie de BA et la raffine avec une stratégie de GA. Pour déterminer quel est le meilleur algorithme de construction parmi ces derniers, on a conduit une série d'expériences qui comparent le temps (mesuré en nombre d'itérations) et la qualité de la solution (mesurée par la longueur de l'ordre, plus la longueur est grande la longueur, plus la qualité diminue). L'évidence empirique suggère que l'HA est la méta-heuristique qui est le mieux adapté pour ce problème en terme de qualité puisqu'elle produit des tableaux plus courts dans un temps raisonnable.

MDCA est construit sur l'hypothèse que l'assurance des interactions inter et intra variable implique que les conditions nécessaires à la prise de décision seront déclenchées. Afin de vérifier ou réfuter cette hypothèse, on a conduit une série d'expériences sur trois sujets d'étude, on compare la couverture sur les conditions des décisions fournies par MDCA, MCA et la génération aléatoire des données.

Les deux premiers sujets d'expérience sont l'exécution d'un même système – système adaptatif de rapport de client (ACRM) [73] – utilisant différentes langues et différents formalismes. Le premier est une implantation utilisant le langage de règles Drools et contient 40 règles. Le deuxième est une implantation utilisant le langage Java et contient 96 règles. Notez que le premier et deuxième sujet n'ont pas un raisonnement équivalent dans tous les cas, les limitations techniques ont mené les développeurs de ces systèmes à prendre des décisions différentes concernant le processus de raisonnement. Le troisième sujet d'étude est un petit serveur web adaptatif implanté avec Java et une langue ad hoc de règle avec un petit environnement composé de trois variables de raisonnement et de 17 règles. Notez que ce système ne prend pas en compte l'histoire de l'environnement pour prendre des décisions.

Les données utilisées pour mener les expériences ont été générées de la manière suivante : l'HA a été utilisée pour produire 50 ensembles de données satisfaisant les propriétés du MDCA (la force et l'enchaînement de la valeur de force égale à 2), un outil tiers [1] a été

1. petit http ://www.mcdowella.demon.co.uk/allPairs.html

employé pour produire 50 ensembles de données satisfaisant les propriétés du MCA , et 50 ensembles de données ont été aléatoirement produits.

L'évidence empirique obtenue à partir des expériences indique que : (i) *MDCA peut effectivement couvrir la plupart des conditions de décision*, (ii) *MDCA surpasse dans les deux premiers sujets la couverture que proposera MCA et la génération aléatoire, et illustre les limitations des MCMA pour couvrir des interactions temporelles*, et (iii) *MDCA et MCM fournissent la même couverture quand le raisonnement ne considère pas l'histoire. Les MCA produisent des tableaux plus courts avec les mêmes résultats que MDCA*. Dans tous les cas, les MCA et le MDCA proposent de meilleurs résultats que la génération de données aléatoires.

Ces observations fournissent assez d'évidences pour affirmer que MDCA assure un niveau de couverture de décisions de raisonnement. Nous comptons qu'avec haut *force* et *force d'enchaînement*, le niveau de couverture augmentera. Cependant, les évidences suggèrent également que MCA et MDCA ont des résultats comparables si le raisonnement ne considère pas l'histoire. Ceci semble raisonnable, puisque MDCA est une prolongation des MCA pour manipuler l'explosion combinatoire produite par l'histoire et les MCA couvrent le même nombre de conditions que MDCA avec moins de données.

0.4.2 Specification des mechanismes orientes aspects

On adresse les problèmes de l'AOP en proposant une caractérisation et un framework de caractéristiques – *Cadre de Spécifications d'Interaction Basé sur les Aspects (ABIS)* [165]. L'idée fondamentale est de fournir aux développeurs et ingénieurs les moyens d'exposer des modules particuliers du système aux aspects, et spécifier les types particuliers d'aspects autorisés pour être tissés avec ces modules.

Le framework ABIS est basé sur une caractérisation des différents patrons d'envahissement que les aspects peuvent réaliser. On identifie onze types d'envahissement selon la façon dont les aspects peuvent modifier le système de base. Notez que cette caractérisation est basée sur la réalisation d'AspectJ de l'AOP et peut être appliqué à n'importe quel langage similaire à AspectJ. Ces patrons d'envahissement sont :

- *augmentation* Le comportement du module intercepté est toujours exécuté, dans ce cas l'aspect augmente le comportement du module.
- *replacement* Le comportement du module intercepté n'est jamais exécuté, dans ce cas l'aspect remplace partiellement ou complètement le comportement du module.
- *conditional replacement* Le comportement du module intercepté n'est pas toujours exécuté, l'aspect remplace le comportement du module du système seulement si une condition est satisfaite.
- *multiple* Le comportement du module intercepté peut-être exécuté plusieurs fois.
- *crossing* L'aspect invoque le comportement de certains modules qu'il n'intercepte pas, cette invocation introduit des dépendances entre modules.
- *write* L'aspect écrit la valeur d'un attribut dans un module, dans ce cas l'aspect modifie les données internes du module.

– *read* L'aspect lit la valeur d'un attribut dans un module, dans ce cas l'aspect accède les données internes du module.
– *argument passing* L'aspect modifie les valeurs des arguments du module intercepté puis il invoque le comportement du module, dans ce cas l'aspect modifie les paramètres de communication entre les modules du système.
– *hierarchy* L'aspect modifie la hiérarchie des modules (héritage)
– *field addition* L'aspect introduit un nouvel attribut dans un module.
– *operation addition* L'aspect introduit une nouvelle opération dans un module.

Il est important de mentionner que les patrons *augmentation*, *remplacement remplacement conditionnel* et *multiple* sont exclusifs, un aspect peut être classifié par seulement l'un d'entre eux.

Étant donné la caractérisation d'aspects, le framework ABIS permet aux développeurs de spécifier les modules du système et indiquer les patrons d'envahissement que l'on autorise ou interdit d'agir sur des modules spécifiques. Par exemple, un développeur peut spécifier dans un module texttttcommunication, que seulement le patron emph augmentation est autorisé d'agir avec ce module. Ceci empêchera d'autres patrons d'agir sur le module `communication`, et permet aux développeurs de commander les aspects qui agissent sur le système de base.

ABIS fournit un langage pour spécifier des interactions permises ou interdites entre aspects dans le système de base. Du côté des aspects, ABIS classifie automatiquement chaque aspect selon le patron d'envahissement qu'il réalise. Cette classification automatique est effectuée en analysant statiquement les éléments structuraux et le comportement de chaque aspect dans le système (avant de les compiler). Du côté du système de base, les développeurs peuvent spécifier les modules du système en utilisant des méta-informations sous forme d'annotations [24]. Puis, ABIS comparera les spécifications du système de base et d'aspect pour vérifier si les deux sont conformes. C'est-à-dire, vérifier que les aspects respectent les spécifications indiquées dans les modules du système de base, et qui ne sont pas tissés dans des modules où ils sont interdits. Dans le cas où les aspects ne respecteraient pas les spécifications du système de base (en se tissant avec des modules erronés), ABIS rapportera des spécifications cassées et en informera les développeurs. Puis, les développeurs pourront corriger le problème et décider si les aspects posent vraiment des problèmes ou si le système de base est le coupable.

On a conduit des expériences sur deux évolutions d'un système orienté aspect. La version originale de ce système est une application chat client-serveur qui met en application des fonctionnalités transversales telles que le chiffrage, jogging, gestion d'erreur, et ainsi de suite en utilisant des aspects (en plus de sa fonction de base des messages à diffusion générale parmi des clients chat). L'évolution du système consiste à modifier le système de base et ajouter des possibilités d'authentification, cette évolution peut être exécutée en employant des aspects ou en modifiant simplement le code du système.

La première expérience a consisté à valider le système et son évolution en utilisant des tests au niveau système et pas de spécifications. Pendant l'évolution, des problèmes de vali-

dation ont été détectés et corrigés en exécutant plusieurs fois les tests et en tissant / défilant les aspects. L'évidence empirique a indiqué que tracer la source des problèmes aux interactions défectueuses d'aspect est un processus pénible, long et complexe qui exige de tisser / défiler des aspects et exécuter des tests plusieurs fois. La deuxième expérience a consisté à ajouter des spécifications d'interactions au système de base et en validant l'évolution utilisant des test au niveau système. Pendant l'évolution, des problèmes de spécifications ont été détectés (des aspects tissées avec de modules trompées) et ont été corrigés sans devoir exécuter tous les tests ou tisser/ défiler des aspects à nouveau. L'évidence empirique a indiqué que quand des spécifications sont présentes, moins d'efforts sont nécessaires pour tracer et corriger des problèmes dus aux interactions défectueuses. On a mesuré les différents efforts pour détecter et corriger les anomalies de la façon suivante : dans la première expérience 55 essais ont été exécutés et 2 aspects ont été tissés/défilés, tandis que dans la deuxième expérience seulement 19 essais ont été exécutés et aucun aspect n'a été tissé/défilé. Ceci implique que les spécifications peuvent effectivement réduire l'effort requis pour diagnostiquer et corriger des problèmes liés aux interactions et à l'envahissement des aspects.

Les spécifications du système de base informent les développeurs au sujet de l'existence des aspects et les forcent à raisonner à l'avance au sujet de tels aspects. Ces spécifications aident aussi les développeurs à gérer le degré d'envahissement des aspects qu'ils permettent dans leur code. Les avantages de ces spécifications sont variés, des nouveaux aspects se tisseront avec le système de base en conformité avec les spécifications. Ceci fait que probablement les aspects capturent moins de modules non souhaités ou qu'ils introduisent des interférences entre aspects. Puisque les intentions d'aspects sont explicites, les développeurs peuvent savoir à l'avance si les aspects présenteront des effets de bord non souhaités ou s'ils interféreront avec d'autres aspects. Si de nouveaux aspects sont introduits mais si l'on peut assurer qu'ils n'ont pas d'effet de bord, alors on est sûr de ne pas exécuter à nouveau chaque test du système.

0.5 D'autres contributions á la validation des systèmes *auto*-adaptatives

Pendant le développement de cette thèse, on a également étudié les défis de l'adaptation conduite par des modèles et d'autres dimensions d'AOP.

D'abord, on a étudié la composition des modèles dans l'adaptation conduite par les modèles. Ces dernières années, les modèles au temps d'exécution et l'adaptation conduite par les modèles ont gagné un grand nombre d'adeptes. La composition des modèles joue un rôle important dans l'adaptation conduite par les modèles. Des modèles sont employés pour séparer les préoccupations métiers et les préoccupations adaptatives, ces modèles sont composés plus tard dans un modèle intégré du système qui est employé pour changer la plate-forme d'exécution. S'assurer que les modèles résultants de la composition sont

comme prévus est important parce que des compositions défectueuses peuvent rapporter des adaptations défectueuses. En [162], on a identifié quelques défis pour la composition des systèmes et modèles adaptatifs, et en [163] on a exploré la validation des moteurs de composition des modèles en proposant un framework de test qui permet aux ingénieurs de tester automatiquement et exhaustivement un moteur de composition des modèles.

En second lieu, on a étudié la diffusion et l'utilisation de l'AOP. Depuis que l'AOP a été présenté en 1997 [126], un grand nombre de chercheurs ont mène leurs recherches autour d'AspectJ [227] – la réalisation la plus populaire de l'AOP. Cependant, il est peu clair si les développeurs emploient l'AOP et comment ils l'emploient. En [166], on conduit une étude empirique sur 38 projets de source ouverte de petite et grand échelle utilisant AspectJ. Cette étude a évalué l'emploi des patrons d'envahissement, la transversale des aspects, les utilisations principales et l'emploi des caractéristiques du langage. L'évidence empirique indique que les aspects sont peu ou pas transversaux, généralement les développeurs de source ouverte n'emploient pas les dispositifs envahissants AOP', et modularisent des préoccupations transversales en utilisant peu d'aspects. La taille du projet n'a pas d'impact sur le nombre d'aspects, et les développeurs emploient juste quelques constructions du langage de point de coupe pour exprimer des aspects.

Ces résultats suggèrent que malgré le grand nombre de travaux et d'avancées scientifiques dans des langages d'aspect, les développeurs sont hésitants pour employer l'AOP. Ceci peut être expliqué par plusieurs raisons : (i) *les développeurs trouvent difficile de raisonner au sujet des unités qui semblent modulaires mais interceptent d'autres unités, en particulier quand ils pensent à AspectJ comme prolongation à OO, qui peut améliorer la modularité mais réduit paradoxalement le maintenance [217]* ; (ii) *Le langage d'AspectJ n'est pas assez flexible pour permettre aux développeurs de modulariser le total des préoccupations transversales* ; (iii) *les caractéristiques envahissantes d'AspectJ, qui devraient aider à modulariser des préoccupations transversales précises ne sont pas employées parce qu'elles peuvent présenter des effets de bord non souhaités [165]*. Cette étude empirique soutient l'intuition que les caractéristiques invasives de l'AOP, l'interférence, et les problèmes d'évolution doivent être abordés afin de soutenir l'adoption de l'AOP en général et en particulier comme mécanisme d'adaptation.

Chapter 1

Introduction

Software systems have propelled the human's race for automation by helping us to automate repetitive yet complex tasks, thus improving the quality of life. Nowadays, we see software systems that help engineers to control nuclear power plants, pilots to fly 500 passenger airplanes, doctors to fetch information of patients in the blink of an eye, and business men to close multimillion deals. These systems play an important role in our society's infrastructure and do more for us today than ever before. Nevertheless, in the future these software systems will provide benefits beyond limits by taking automation to the next level.

The next generation of software systems should answer the needs of the dynamically changing world in which we live [134]. Future software systems are expected to operate non-stop 24/7 for long periods and offer flexibility, reliability, and robustness. To support these characteristics, these systems need to have the ability to change their internal structure and behavior, and judge when such changes are needed. These systems are called *self*-adaptive and are capable of delivering a dramatic improvement in applications domains such as crisis management [114, 112], space exploration [72], home-automation [157, 170, 160], and business applications [27].

1.1 *Self*-adaptive systems

Self-adaptive systems are the future of software automation [43] and should improve the automation offered by current software systems.

Customer relationship management systems (CRM) [3] exemplify such improvement. Classical CRMs are business applications that automate and centralize the handling and the development of customer relations. CMRs enable executives to do activities such as handle appointment, share files, retrieve and modify clients' information, and send automated messages. Today, CRMs [120, 3] propose a variety of operation modes according the different operational environments. Each version of a CRM provides access to a variety

of calendar services, encryption protocols, and user interfaces. Then, executives should choose the version that best fits their needs, turn off the current version, and start the new one. This renders the use of CRM systems across different operational environments complex and tedious.

Adaptive CRM systems (ACRM) [73] are *self*-adaptive systems that improve the automation of classic CRMs. ACRMs provide the same functionalities provided by CRMs, but offer dynamic and autonomous variability. These systems are capable of self-modification. This allows them to change dynamically and autonomously components such as calendar services, encryption protocols, messaging facilities, and user interface. Such changes occur according to the environment needs, on the fly and without needing any restart or intervention. ACRM systems improve the automation and the benefits that classic CRMs provide. ACRMs relieve executives from the burden of selecting the right CRM or picking and using one that is unfitted to their needs.

The anatomy of *self*-adaptive systems is the following [111]: (i) *Self-adaptive systems sense a set of relevant properties from the working environment*. For example, an ACRM senses the network connection speed, network security level, and available calendar service change. (ii) *Self-adaptive systems reason and make decisions based on these environmental property values*. For example, when the connection speed is low, the security level is high, and the Google calendar service is available, it will decide to make the Google calendar client available. (iii) *Self-adaptive systems apply the decisions that modify their underlying structure*. For example, following the decision to make available the Google calendar client, the ACRM disconnects the previous calendar modules, and connects the Google calendar module, the Google messaging service module, and the Google calendar user interface.

Designing and building *self*-adaptive systems is not simple [44] and poses several research and engineering challenges. Researchers and engineers need to create several pieces of technology such as techniques to sense the environment [42, 81, 18], technology to make decisions based on innumerable environmental conditions [91, 105, 125], and mechanisms that enable adaptation (reconfiguration) on the fly [31, 220, 156]. Each piece of these pieces embodies an overwhelming complexity [44]. Engineers building *self*-adaptive systems are prone to make a series of mistakes and to introduce faults into these systems.

Faults in *self*-adaptive systems may have minor consequences such as rendering the access to calendar services impossible, to more serious consequences such as disabling the engine control modules of a flying airplane. Such faults could be found in each component that integrates a *self*-adaptive system. Engineers could create a wrong representation of the environment and environmental properties, reasoners that make wrong decisions, or adaptation mechanisms that apply incorrectly the adaptations.

Self-adaptive systems can effectively improve automation in a large number of human activities. Nevertheless, our society will benefit of such automation only when engineers could guarantee that these systems will perform as expected. This implies that *self*-adaptive systems must be tested and verified thoroughly to find and correct faults, and to guarantee that each piece of these systems is valid and works correctly.

1.2 Challenges of the validation of *self*-adaptive systems

The validation of self-adaptive systems consists in guaranteeing that each component of these systems functions correctly [30]. To achieve such validation, each piece that constitutes the system must be validated thoroughly. Nevertheless, the validation of each of these pieces carries a series of challenges that need to be conquered.

The validation of sensors and environment models consists in ensuring that the environment is well represented by the environment's models and that probes can effectively sense the environmental variations [44, 30]. The challenges associated with the validation of the environment representations lie on the complexity introduced by environmental properties, which can be continuous or discrete, by environments that may behave erratically, and by sensors that may provide erroneous data making the environmental models inconsistent [230].

The validation of reasoning engines – the pieces of software entitled with the decision making process – consists in ensuring that they make the right decisions given any of the possible environmental conditions to which the system is intended to adapt [44, 30]. The challenges associated with the validation of reasoning engines relate to the large number of possible environmental conditions that a self-adaptive system will face. The number of environmental conditions grows exponentially with the number of properties that drive the adaptation. Furthermore, reasoning engines may make decisions based on previous environmental conditions that span over a time window. Since these environmental conditions represent temporality, the order in which they occur is important. Therefore, the number of possible arrangements in which environmental conditions may occurs must also be considered. Yet, the number of possible arrangement for environmental conditions grows exponentially with the size of the time window. Validation techniques for reasoning engines must be capable of handling a huge number of environmental conditions and possible arrangements of them.

The validation of adaptation mechanisms – the pieces of software entitled with the dynamic adaptation – consists in ensuring that the system will reconfigure correctly and that new reconfigurations will not break the system down [44, 30]. This requires ensuring that every possible variant of the system is feasible, that they will not break down the system, and that they will provide the functionalities they are meant to. Additionally, validation should ensure that the system's components that enable dynamic change do not interfere with each other. The challenges associated with the validation of adaptation mechanisms relate to the large number of components, interactions between components, and possible system variations. Furthermore, different adaptation mechanisms may introduce different limitations or advantages to the validation. For example, adaptation mechanisms based on aspect-oriented programming allow engineers to master the large number of possible system variants [220], but introduce a series of challenges related to unintended interactions between modules and undesired side effects [150, 121]. Validation techniques for adaptation mechanisms must be capable of handling a large number of components and

interactions, and provide assurance that new reconfigurations will be safe.

1.3 Contributions of this thesis

In this thesis I present two major contributions that address respectively the validation of reasoning engines through testing and the validation of adaptation mechanisms based on aspect-oriented programming through specifications.

1.3.1 Test data selection from reasoning engines

The first contribution of this thesis addresses the validation of reasoning engines through testing.

Software testing is an empirical validation technique, which focuses on the execution of the system realization and the evaluation of the observable results produced by the system. Testing reasoning engines consists in feeding the system with a set of environmental conditions, and then evaluating the resulting decisions made by the reasoning engine. If the resulting decision is as expected, then the system is valid, otherwise it contains faults that need to be found and corrected. Ideally, engineers should test reasoning engines against every possible environmental condition and the different combinations of them. Yet, as mentioned earlier, the number of environmental conditions is huge, and testing all of them is most often impossible.

In this thesis, I introduce a test data selection technique, which allows engineers to sample relevant environmental conditions and interactions among these conditions. In this way, engineers do not need to test every possible environmental condition, but a limited number of them in order to validate reasoning engines. This technique, called *multi-dimensional covering arrays* (MDCA) is capable of handling a very large number of environmental conditions by sampling the ones that are more likely to uncover faults. The idea underlying MDCA is the selection of (i) *relevant combinations (combinatorial selection) of the environmental property values that constitute each environmental conditions*, and (ii) *relevant interactions among environmental conditions that represent the temporality*. For example, MDCA may select all the pairs of environmental property values to construct environmental conditions, and all the arrangements of two environmental conditions to reflect the temporality of environments (in a time window of two environmental conditions).

MDCA aims at making a compromise between testing a large number of environmental conditions and selecting a limited amount of them without sacrificing the capability to find faults. MDCA can effectively trigger faults in the reasoning process by selecting environmental conditions that satisfy a combinatorial criterion. It supports engineers in the validation of reasoning engines by allowing them to select environmental conditions that are representative of the environment and its variations. These environmental conditions can be then to test, detect, and find faults in reasoning engines.

The contributions of this thesis to the test of reasoning engine are: (i)*a formal definition of MDCA*, (ii) *a set of techniques to construct MDCAs*, and (iii) *an empirical validation of the MDCA's capacity to cover reasoning engine's decision*, which should lead to the discovery of faults.

1.3.2 Specification of aspect-oriented adaptation mechanism

The second contribution of this thesis addresses the validation of aspect-oriented adaptation mechanisms through specifications.

Aspect-oriented programming (AOP) was introduced in 1997 [126] as a way to modularize concerns that crosscut several modules in a software system. AOP provides the means to encapsulate into well modularized units of code – *aspects* – those concerns that are spread across several parts of the system. Such concerns may modify the underlying structure and behavior of the base system (system without aspects). Furthermore, AOP provides the means to dynamically weave and unweave those concerns from the base system [184]. These characteristics make AOP a good candidate to perform the adaptations in a self-adaptive system. An aspect-oriented adaptation mechanism may use aspects to realize the adaptive concerns (those that modify the system structure and behavior) and dynamic weaving / unweaving mechanisms to perform the system's adaptations [151].

Nevertheless, we recently observed that AOP has experimented a slow adoption in the last years [166]. This slow adoption is due to three factors, (i) *the evolution and maintenance problems of AOP* [150], (ii) *the uncontrolled and unexpected interactions among aspects* [121], and (iii) *the uncontrolled invasiveness of aspects over the base system* [80, 164]. These three factors represent major difficulties for the validation of aspect-oriented programs [165], and particularly impact the validation of aspect-oriented adaptation mechanisms. Aspects realizing adaptive concerns may introduce unexpected interactions, undesired side effects, and unforeseen dependencies with the base system. Engineers validating adaptation mechanisms are forced to test every possible configuration (weaving) because there is no assurance that the addition or removal of an adaptation concern (aspect) will not introduce side effects. Nevertheless, as mentioned before, due to the large number of adaptive concerns and possible system variants this is often impossible.

In this thesis, I introduce an interaction specification framework [165] – *ABIS* – to control the interactions between aspects (adaptive concerns) and the base system. ABIS enables engineers to specify the type of aspects (which carry adaptive concerns) that they allow or forbid to be woven with the base system. These aspects are identified by the *invasiveness pattern* they realize, thus providing a mechanism to control the invasiveness of aspects. ABIS provides the means for engineers to ensure that aspects interacting with the base code will not introduce undesired side effects. If new adaptations are introduced into the system developers do not need to retest all the system functionality. Additionally, in the case where aspects introduce faults, ABIS can provide useful information and reduce the effort needed to find and fix those faults.

Invasiveness patterns are a characterization of the invasive capabilities of aspects. These patterns classify aspects among 11 types of invasiveness ranging from control flow invasiveness, to data and structure invasiveness. ABIS is capable of automatically classify aspects according to invasiveness pattern they instantiate in their code.

The contribution of this thesis to the validation and specification of aspect-oriented adaptation mechanisms are: (i) *a characterization of invasiveness patterns for AspectJ*, (ii) *an automated classification technique for invasiveness patterns*, (iii) tooling support for the ABIS specification framework, and (iv) *an empirical demonstration of the overall benefits of specifying the interaction between aspects and the base system*.

1.4 Organization of this thesis

This thesis is organized as follows. In chapter two I present the background (state of the art) and motivation of this thesis. In chapter three, I introduce a test data selection technique, MDCA that targets very large environments in which temporal variations are meaningful. In chapter four, I introduce an interaction specification framework for AOP, ABIS, which is intended to provide assurance about the interactions between aspects and the base system. In chapter five, I conclude and discuss the perspectives of this thesis.

Chapter 2

Background and Motivation

Researches have invested major efforts in developing techniques and methodologies for developing *self*-adaptive systems from top to bottom. These techniques range from environmental data acquisition, reasoning techniques, platform reconfiguration, and model driven methodologies to adaptive validation and verification techniques for *self*-adaptive systems.

In this chapter, I survey these efforts: First (Section 1), I survey the fundamental pieces and the techniques that researchers have proposed to build and manage *self*-adaptive systems; Second (Section 2), I survey the software testing and verification techniques for these systems; Third (Section 3), I position this thesis in relation to *self*-adaptive systems and existing validation ‡ and verification (V&V) techniques related to self-adaptation.

2.1 Self-Adaptive Systems

> *– Cordelia is a high ranked executive in a big business consultant firm. Her day-to-day duties range from presiding customer satisfaction meetings to visiting customers in the homeland or overseas. Cordelia is constantly moving, changing from one office to another, visiting customers, having meetings, and so on. Her working environment is constantly changing and heterogeneous. Her daily work relies on a particular information system, which manages customer's information, corporative address book, email, calendar, and Smart phone facilities.*
>
> *She heard long ago that this system is called ACRM, which stands for Adaptive Customer Relationship Management. What's more, according to the technicians she usually speaks to, the system is capable of changing itself to satisfy her unique requirements. It's like having a special version of the system for her – Cordelia's ACRM.*

‡. See the glossary in Chapter C.

Self-adaptive systems answer a need to manage the complexity and the requirements of changing operational contexts and environments [43]. For example, in the previous excerpt, Cordelia's ACRM [73] is a *self*-adaptive system. It is meant to: (i) *execute for very long periods (sometimes even eternally)*, and (ii) *continue functioning and providing the better service as possible when its working environment changes*. This kind of systems must change their internal structure and properties to keep going over time. Through *self*-adaptation these systems offer versatility†, flexibility†, resiliency†, dependability†, robustness†, recoverability†, customizability†, and so on.

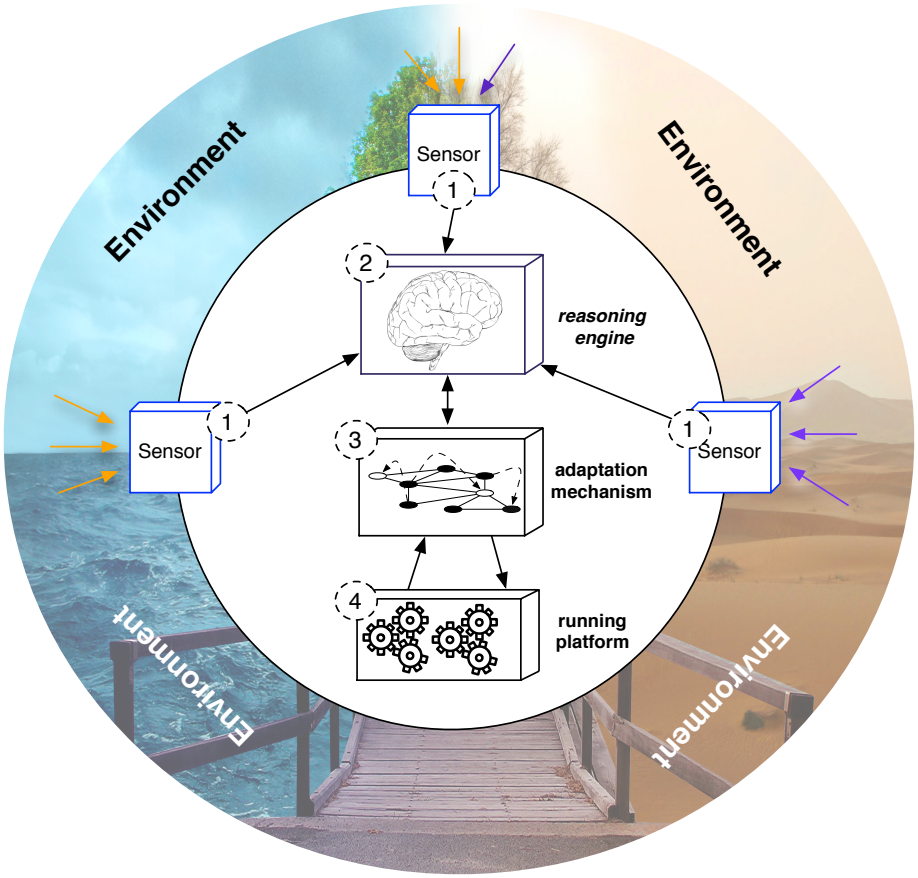

Figure 2.1: *Self*-adaptive system

Figure 2.1 illustrates the main *self*-adaptive components and their interactions. In general, a *self*-adaptive system is composed of four bricks [111]. The first consists of a series of *sensors* (1) or *monitors* that detect the environment's fluctuations. These sensors translate the fluctuations into a *representation* of the environment, which is then interpreted by a *reasoning engine*. The second is a *reasoning engine* (2) or decision maker. It controls the re-

lationship between environmental variations and internal changes in the system structure and behavior. Such relationship is expressed in the form of decisions to perform atomic actions that constitute an overall change in the system's structure and behavior. Finally, the third and fourth parts are an *adaptation* (3) or reconfiguration mechanism and a *running platform* (4). The running platform is the system's executing structure that provides functionalities, whereas the adaptation mechanism is the executor of the decisions (actions) made by the reasoning engine. The adaptation mechanism is responsible of mutating the running platform in the right way according to the reasoning engine orders.

Each of these *self*-adaptive bricks can take a different shape depending on the particular needs of the underlying application domain. The representation, measurement, and monitoring of the environment may change according to the nature of the underlying physical environment (if any). Reasoning engines may vary according to the different environment representations and the purpose of the adaptations. Some application domains may promote using rule based systems, whereas others may promote using utility function based adaptation. Analogously, adaptation mechanisms may vary according to the nature of the running platform. Some mechanisms may use hardcoded and predefined adaptation actions, whereas others may use more abstract facilities such as models that represent the system structure.

In this section, I survey these bricks and provide an overview of a wide range of options and techniques to realize each of them. Notice that the acquisition and monitoring of environment is outside of the scope of this thesis, and therefore I do not survey the work in this field. Abide readers willing to know more advance about that particular field can refer to [111].

2.1.1 Environment representation

> *– Cordelia is a moving executive. She is always on the road, visiting customers, moving from one office to another, changing from smart phones to laptops and desktop computers. Sometimes, she can only access the 3G data network, and sometimes she has fast internet access. At work she has a secure network connection, whereas at home she has an insecure connection. Cordelia intensively uses her calendar interface to add, modify, and view appointments. Her working environment is intensively nomadic. Cordelia's Customer relationship management system must adapt to these environments as best as possible.*

Environments and environmental conditions can be modeled with a variety of techniques [106, 230, 42, 81] that share a set of common representation elements. These commonalities represent environmental properties as variables with well defined discrete domains. For example, the environment described in the previous excerpt is represented by a set of *reasoning variables* such as *connection speed* (network connection speed). Each variable conveys a value, which is representative of the current environment state. For

Table 2.1: Reasoning variables and their domain

Variable Name	**Variable Domain**
exchange service	*available*, *unavailable*
Googleservice	*available*, *unavailable*
ical service	*available*, *unavailable*
groupwise service	*available*, *unavailable*
oracle service	*available*, *unavailable*
memory level	*low*, *high*
usetype	*view*, *reserve*
platform	*windows*, *mac*, *palm*, *mobile windows*
security level	*low*, *medium*, *high*
user feedback	*like*, *dislike*
connection speed	*slow*, *fast*

example, *connection speed* is a reasoning variable whose value can be either *slow* or *fast*. Table 2.1 summarizes the reasoning variables for the Cordelia's ACRM system.

Reasoning variable values represent the state of a particular environmental property at a precise instant, and a vector of reasoning variable values represent the state of the environment at a precise instant.

Definition 1. *An environment instance (instance for short) is a vector of reasoning variable values that represent the state of the environment at a precise instant.*

In the literature, some authors [106, 230, 42, 81] refer to the environment instances as *contexts* or *environments*. Since the environment is continuously changing, the system perceives it as a sequence of *instances*. Table 2.2 presents two instances of the environment of Cordelia's ACRM. Each row contains a reasoning variable value, and each column contains an instance. The relation between context instances is described by *instance transitions*.

Definition 2. *An instance transition (transition for short) is a pair of ordered environment instances that represent the temporality of the system.*

For example, the change from instance 1 to instance 2 in Table 2.2 is a transition. In a sequence of instances, the reasoning variables may change their values several times. I refer to the number of changing values as *change rate*.

Definition 3. *A change rate n indicates number of reasoning variable values that variate in a sequence of instances.*

For example, when the reasoning variables changes from *low* to *high*, it has a change rate 2 because it involves a change between two values. A change rate 3 means that a reasoning variable consecutively changes its value twice between three values. Notice

that there is a direct relation between the amount of transitions and the change rate of a reasoning variable. For a variable with change rate 2, two instances are needed, with change rate 3, three instances are needed, etc.

Table 2.2: Reasoning var. values for two *instances*

Var. Name	**Inst. 1**	**Inst. 2**
exchange service	*available*	*available*
Googleservice	*available*	*unavailable*
ical service	*unavailable*	*available*
groupwise service	*unavailable*	*unavailable*
oracle service	*unavailable*	*available*
memory level	*low*	*high*
usetype	*view*	*reserve*
platform	*palm*	*mac*
security level	*medium*	*low*
user feedback	*dislike*	*dislike*
connection speed	*slow*	*fast*

Another environment representation consists in using software product lines [48] to model the environment† and its variability [18]. The underlying idea is to model the environment as a software product line, which represents the different values of environmental properties as options or variation points. Then, a product of the product line represents a particular environmental condition (instance). This product is the result of selecting particular variation points for the product line. Figure 2.2 illustrates a product line feature diagram for Cordelia's ACRM. On the top of the hierarchy we find the environment, which contains the different environmental properties (reasoning variables, gray nodes in the figure). The leaves in the hierarchy represent the possible environmental properties' values.

Notice that the each representation models the available service property in a different way (cf. Table 2.1 and Figure 2.2). In one representation, reasoning variables can have only one value, whereas in the second a reasoning variable can have multiple values with a defined cardinality. For example, in Figure 2.2 the available services are *exchange* and *google*, and the reasoning variable *service available* contains these values. In contrast, in Table 2.1 the same services are represented by the value *available* of the reasoning variables *exchange service* and *Googleservice*.

Both representations are similar and equivalent, however, the first supersedes the second. A software product line can be represented as properties and values with some constraints attached to them. Yet, the second aims at giving a clear view of the environment through a hierarchical representation of the environment concepts. Contrarily, the first is flat and provides a simpler vision of the environment, which can result being cluttered on

environments with many reasoning variables.

The number of possible environmental instances can be huge according to the selected environment representation. For example, consider the environment as modeled in Table 2.1. The number of possible environment instances is equivalent to the number of ways in which it is possible to select a particular value for each reasoning variable. Then, there are $2^9 \times 3 \times 4 = 6.144$ **instances**, and $6.144^2 = 37 \times 10^6$ **transitions** containing variables with change rate 2.

These numbers are relevant because the system must reason on that number of environment conditions and define a mapping between the possible environmental conditions and the different reconfigurations it can perform. **In general, the bigger is the environment, the more complex is the reasoning**. In the next section I survey a number of reasoning strategies and I show how environment instances and transitions related to the reasoning process.

2.1.2 Autonomous reasoning

> *– Cordelia opens her smart-phone and checks the next meeting with customers. She notices that her device is running out of memory and that her preferred calendar is not available in the mobile platform. She turns off her phone and as soon as she opens her laptop, the calendar client pops-up announcing that the next meeting is about to start.*

Domain experts know how the system should respond given a change in the execution environment. For example, a domain expert knows which calendar to select given a particular set of services, connection speed, and other values for environmental properties. In adaptive systems, the reasoning process encodes the domain knowledge into mapping between the different environmental conditions and the system reconfigurations (actions) [69, 161, 205].

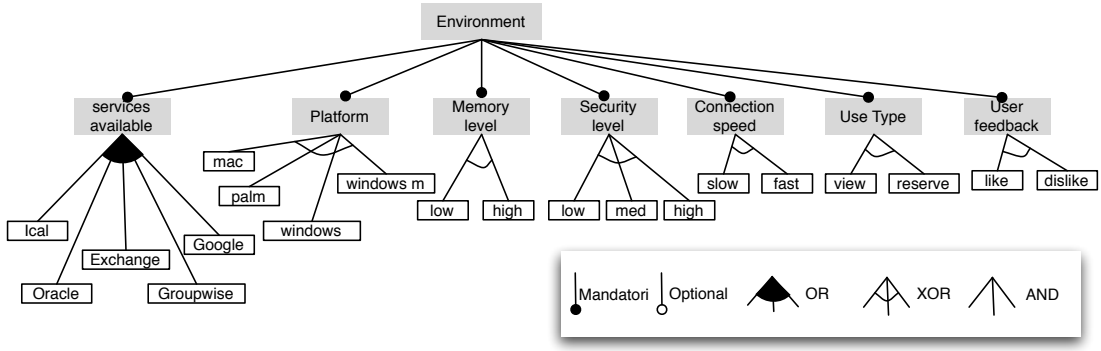

Figure 2.2: Product line variability model for Cordelia's ACRM environment.

The reasoning process is typically entitled to a piece of software known as *reasoning engine*. Typically, a reasoning engine link particular reasoning variable values with predefined actions that lead to reconfigurations of the underlying running system. Reasoning engines operate in two steps [69] [1]: (i) *they process a sequence of environment instances*; and, (ii) *they make decisions about the action to perform given an environment instance*.

For example, in the previous excerpt when Cordelia switched from her mobile device to her laptop computer an environmental change happened. The reasoning engine of Cordelia's ACRM processed this change, and based on the environmental information it decided to engage another calendar client and activate the *pop-up* notification module.

> *– The system pops-up a calendar that Cordelia really doesn't like, it is something about the single-touch interface that really disturbs her. After a few times of receiving her feedback, the system learns that Cordelia doesn't like that particular calendar, then the calendar doesn't pop-up any more.*

Reasoning engines are not limited to make decision on punctual variable values, they can also reason over streams or sequences of environment instances. Past environmental conditions may be used to reason about new environmental conditions [146]. For example, in the previous excerpt, Cordelia's ACRM system learns that she does not like a particular calendar client interface. This is possible because the reasoning engine receives and processes Cordelia's feedback (in the form of environmental data), and after a pre-defined number of times that she indicates her reluctance to a calendar client, the system creates new knowledge that forbid the selection of that calendar. In this case, the reasoning engine is using historical environmental data to refine its reasoning.

Other form of history-based reasoning consists in taking into account just the one (or more) environment instance back in history [61, 219]. For example, when the platform changes from *palm* (past environment instance) to *mobile windows* (current environment instance), only two out of ten calendar clients can be used. Notice that the state (configuration) of the system can also influence the reasoning process and may in most cases be considered by the reasoning engine.

There exist a variety of techniques to build reasoning engines and capture domain knowledge. Three of these techniques *Rule based reasoning* [91], *Goal based reasoning* [74], and *Utility function reasoning* [83] are mature enough to be used by most *self*-adaptive systems, whereas other techniques [128, 35, 75, 107] are still in early research stage.

In this section I survey rule, goal, and function based reasoning, and I present an overview of other reasoning techniques. Notice that in this thesis I focus particularly on systems using rule based reasoning, however, I aim at covering as many reasoning techniques as possible.

1. Dobson et *al*. [69] define four steps for reasoning collect, analyze, decide, and act. In this thesis, the collect step is embodied in the environment representation, the act step is embodied in the adaptation mechanism, and the the analyze and decide steps are performed by the reasoning engine.

Rule Based Reasoning

Rule based reasoning is founded in expert and knowledge based systems [84, 85], which emerged in the early 60's as a way to emulate the expert human thinking. Nowadays we see rule based system in several areas of knowledge such as assisted legal systems [235], business process [189], and adaptive systems [91].

Rule based reasoning consists in making decisions given a set of events (such as values for reasoning variables). Typically, rules are expressed in the form of *if (condition)-then (action)* statements [91, 123, 81]. When a set of events satisfies the *condition*, then the *action* is triggered. Some variants of rule based reasoning, usually named event-condition-action (ECA), consider that *event* may trigger rules containing *conditions*, that if satisfied activate a set of *actions* [156, 40, 149, 92]. In general, the conditions are punctual reasoning variable values. However, some approaches prefer to use qualitative descriptions instead of punctual values [40, 41], this is particularly useful when dealing with discrete reasoning variable values. In this kind of rules, variable values qualifications such as *good* or *bad* may vary according to other variable values. Such rules handle imprecision and variation using fuzzy logic [234].

Several languages have been proposed to realize rule based reasoning. Functional programming, particularly LISP has been a source for rule languages such as CLIPS and Jess [108]. Declarative languages such as Ponder [61] and Ponder II [219] allow developers to write ECA rules using a textual syntax. Commercial initiatives such as the Microsoft BizTalk [130] product allow developer to write rules in a *graphical* way. The Drools language [116], a rule language produced by JBoss (a division of Redhat Inc.), provides both graphical support and text syntax to write declarative rules. JRules [113] is another rule language similar to Drools produced by the IBM Corporation. Notice that Jess, Ponder I & II, and Drools are well integrated with the object-oriented language Java™. Notice that these rule languages (with the exception of Drools) can only reason about punctual events at each time (they can however, aggregate several facts using logic operators such as *and* and *or*).

```
1 rule "R1: platform windows−reserve"
2     when
3         Platform( id == Platform.WINDOWS or
4                   id == Platform.MAC )
5         UseType( type == UseType.RESERVE )
6     then
7         ORACLEClient.setActive(true);
8 end
```

Listing 2.1: Reasoning rule for the *exchange service* availability

Listing 2.1, illustrates how the first excerpt of the previous chapter would be expressed in Drools expert language. The rule *platform windows-reserve* is composed of two parts, *conditions* and *actions*. The construct on lines 3 and 4 declare the conditions that will trigger the rule. The expressions *Platform* and *UseType* correspond to reasoning variable types. The expressions inside parenthesis are conditions on type attributes. The condition in lines 3 to 5 requires the attribute *id* of a variable of type *Platform* to have a value equivalent to the constant *Platform.WINDOWS* or *Platform.MAC*. The construct on line 7 is the actions to perform when the conditions are satisfied. In this case, the action consists in setting the attribute *active* of the constant *ORACLEClient*, which represent actions that the reasoning engine can use to make a final decision. Therefore, this rule triggers only when the current platform is *windows* or *mac* and the use type is *reserve*, and activates the calendar client *oracle* to be used.

A particular kind of rule based reasoning natively considers the notion of temporal reasoning (or reasoning about historic facts) [146]. That is, the rules declare conditions over past facts (or sequences of them), and define an explicit representation of temporality (before, after, while, etc.). Most rule languages do not natively support the expression of this kind of rules. Those languages that support the interpretation of past facts are called *complex event processors* [146]. Among such languages we find WildCat [63, 1], an extensible Java framework that provides a query language to process complex events; and, the Drools language [116] that provides both complex event processing engine [2] and rules on punctual events. Other languages [108, 61, 219, 130] need ad hoc hacks and tricky rule manipulation in order to include past facts in their decisions.

```
1  rule "R2: security−protocol transition"
2      when
3          $currentServices :
4              Services( available contains Services.Google)
5          Services( this before $currentServices,
6              selected contains Services.EXCHANGE)
7          $currentSecurity :
8              Security( level == Security.LOW )
9          Security( this before $currentSecurity, level == Security.HIGH )
10     then
11         TLSProtocol.setActive(true);
12 end
```

Listing 2.2: Reasoning rule considering history

2. The drools language comprises two main functionalities, rule processing (Drools expert) and complex event processing (Drools fusion). Both functionalities can interoperate and trigger decisions /actions.

Listing 2.2, illustrates a rule handling temporal conditions. The rule *security-protocol transition* detects when the environment changes from service available *exchange* to *google*, and from security level *high* to *low*. The particularities of temporal conditions are noted in lines 3 to 9. The keyword this indicates the current variable which satisfied the conditions after the comma separator. The keyword before (as well as after) refers explicitly to a temporal occurrence of the variable. In lines 3 and 4, the local variable *$currentService* (line 3) holds the value of a reasoning variable representing the Googleservice (line 4). In line 5, the keywords this and before are used to establish that a condition exchange service is available before the Googleservice.

Rule based reasoning engines can make use of historical data to create or modify existing rules. This is particularly useful when new reasoning knowledge is derived from past facts [58, 67]. There are certain rules that can create, modify, or delete other rules, these are called *meta-rules*. In general, there is no particular rule language that supports this kind of rules. However, the only difference between meta-rules and regular rules are the actions that modify and create new rules. Some languages, such as Drools and Ponder, provide the support to create ad hoc hardcoded infrastructure to support the expression of meta-rules.

```
1  rule "R3: user feedback"
2      when
3          $feed : UserFeedback( feedback == UserFeedback.DISLIKE )
4          $client : CalendarClient( state == CalendarClient.ACTIVE )
5          Number( intValue > 10 ) from accumulate (
6              CalendarClientFeed( client == $client, feed == $feed )
7              over window:length( infinite ), sum(1))
8      then
9          SystemManager.createNewRule(INACTIVE,$client);
10 end
```

Listing 2.3: Reasoning rule considering a chain of events

Listing 2.3, illustrates a meta-rule that realizes the behavior described in the second excerpt of the previous section. The meta-rule *user feedback* detects when the user feedbacks the system more than ten times about his dislike with respect to a particular calendar client interface. Then, the system creates a new rule that whenever possible avoids the selection of the disliked calendar interface. The condition in lines 5 to 7 contains a temporal condition, which counts the negative user feedback associated with each available calendar over an infinite time window. The action of this meta-rule is encoded as a support function written in Java and creates new rules dynamically and feeds them into the reasoning engine.

Writing reasoning rules appeals to a situational thinking and therefore it appears to be very intuitive. Nevertheless, when writing reasoning rules it is hard to foresee when rules interaction may introduce errors in the reasoning [149, 205]. Furthermore, it is difficult to reason about history and previous events using rules because the occurrence of such events must be explicitly stated in each rule.

Goal Based Reasoning

Goal based reasoning is a technique inspired by multi-agent planning systems and particularly *belief-desire-intention* (BDI) reasoning [190]. In BDI a set of software / robotic agents perform a set of tasks that drives them to reach a pre-defined goal, for example getting from a point A to a point B while skipping obstacles.

Goal based reasoning works in the following way. High-level goals drive the adaptation process. Many goals can be expressed and these can interfere with each other [124]. Examples of goals are *there is always an active calendar client* or *ensure that the user is satisfied with the calendar client selected by the system*. These goals are expressed using a declarative language such as GOAL [110], GOLOG [141, 88], PROLOG [29], etc. Particular actions such as *changing the calendar client*, *changing encryption protocol* and so on must also be defined in a declarative way. When the environment changes, a set of rules decides if the system is still accomplishing its goals. If not, the reasoning engine selects among a set of reconfiguration (sequences of actions, or reconfiguration plan) intended to lead to the goal. Typically these plans are known as reactive plans [169].

Several researchers have explored goal based reasoning in *self*-adaptive systems [74, 105, 154, 213, 214, 124, 195]. Usually, they define the actions to include in the adaptation plans in terms of atomic actions such as removing software components, changing software parameters, or adding new components. High-level goals are then expressed in terms of particular properties the system should have. For example, *always selecting a calendar client given an environmental change*. Processes are important pieces that connect the set of atomic actions and goals. Processes define, for example, the order in which two or more actions must be applied given a particular decision, or environmental change.

```
1 primitive_action(disable_calendarClient(C)).
2 primitive_action(activate_calendarClient(C)).
3 primitive_action(keep_calendarClient(C)).
4 ...
5
6 proc(env_cal_service(E,L), google_service(E,L);
7      exchange_service(E,L); ...)
8 proc(swap(C1,C2), ? current_cal(C1)
9      # disable_calendarClient(C1)
```

```
10      # enable_calendarClient(C2))
11  ...
12  /* goal 'always keeping a calendar client active'*/
13
14  holds(active_calendar(C),E) :- current_cal(C).
15
16  /* E environment, L list of calendar clients, A action, S */
17
18  holds(on(E),do(A,[C|L])) :-
19      A = activate_calendar(C);
20      A = disable_calendar(C);
21      A = keep_calendarClient(C);
22      not A = activate_calendar(C),
23      not A = disable_calendar(C),
24      not A = keep_calendarClient(C)
25      , holds(active_calendar(C),E);
26      do(A,L).
```

Listing 2.4: Goal based reasoning in GOLOG

Listing 2.4, illustrates a partial excerpt of a goal based reasoning for Cordelia's ACRM in the GOLOG language. The reasoning is intended to lead to the accomplishment of the goal *there is always an active calendar client*. To do so, a set of actions is declared using the *primitive_actions* keyword (lines 1 to 4). These actions are meant to reconfigure or modify the system. A set of *action controls* defined with the *proc* (lines 6 to 11) keyword rule the link between actions, environmental conditions, and goals. High-level goals are codified in the form of logical assertions stating properties that the reasoning must hold. When the environment changes the reasoning engine computes a reconfiguration plan (sequence of primitive actions) that will lead the system to achieve the goal.

Writing high level goals in a declarative way appears to be counter intuitive because even simple goals may span to several lines of code. This motivated researchers to propose expressing high level goals using linear temporal logic [213, 214] to then automatically generate reactive plans that follow these goals. However, primitive actions and the processes linking goals and actions still need to be written manually in a declarative language. Regarding temporal reasoning, goal based systems can easily incorporate temporal assertions using logic statements (constructs such as *before* and *after* can easily be expressed in this way).

Goal based reasoning appeals to a goal directed reasoning, where the goal primes over the means to achieve that goal [84]. This is also the main benefit of goals based reasoning, through the establishment of high-level goals, it avoids the need of complex situational reasoning. On the other hand, the main difficulty with this kind of reasoning is that writing

goals, actions, and the logical statements that allow deriving a reactive plan requires a particular thinking and mental plasticity. Besides, it is notably difficult to predict when multiple goals will prevent the system to reach a goal at all or if all the goals can be attained.

Utility Function Reasoning

Utility function based reasoning is a reasoning technique rooted in the concepts of functional optimization [232]. Functional optimization consists in given a continuous or discrete function (expressed in mathematical terms), find its optimal value with respect to a set of parameters.

In *self*-adaptive system, utility function based reasoning uses functional optimization in order to select the best-fitted system configuration to answer to the environment needs. The function to optimize is called *utility function*, and is defined in terms of the utility or service that each configuration can provide given particular environmental conditions. More precisely, a utility function defines a quantitative level of *desirability* to each reconfiguration. This desirability is a mapping between each configuration and its worthiness with respect to environmental conditions. Then, when the environment changes, the reconfiguration to apply (decision to make) is the one that maximizes (or minimizes) the utility. Several methods can be applied to search the optimal value of these functions varying from linear programming to meta-heuristic search [232].

Several researchers have investigated the application of this technique to decision making in *self*-adaptive systems [83, 125, 168, 199, 178, 45]. Usually researchers define the utility function as the sum of the utility value of each moving part of the system (reconfigurable elements of the platform) in relation to the environment. This relation is predefined as a normalized mapping (function) between each reasoning variable value and the system configuration. Additionally, the utility function may consider penalties for undesirable configurations. This requires foreseeing the possible system configurations and environments to encode their relation in a mathematical function that expresses the best possible reconfiguration (change).

For example, Cordelia's ACRM system is composed of three reconfigurable parts. A calendar client (CC), which can vary according to the available calendar server; an encryption protocol (EC), which can change according to the available connections speed, security level, etc; and a messaging system (MS), which may vary according to the underlying *platform*, *security level*, etc. These three elements constitute the system configuration *conf* = {*CC, EC, MS*}. The utility value of this configuration is defined by a function that maps its values (for example, *conf* = {*outlook, ssl, sms*}) to quantitative values. In this case, the utility function $U(conf)$ is defined as the sum of the contribution that the configuration

can make per reasoning variable. The Utility function for Cordelia's ACRM is:

$$U(conf) = \sum_{i}^{reasoning\ vars} U_i(conf) \tag{2.1}$$

$$\begin{aligned} U(conf) = {} & U_{service}(conf) + U_{memory}(conf) + U_{usetype}(conf) \\ & + U_{platform}(conf) + U_{security}(conf) + U_{feedback}(conf) \\ & + U_{connection}(conf) \end{aligned} \tag{2.2}$$

This function cumulates the worthiness of each configuration in relation to the reasoning variables. Typically, the utility is a discrete function since it is not always possible to translate the configuration contributions to a continuous domain. The different reasoning variable values are encoded inside each variable function. For example, the function for the reasoning variable *memory*, encodes the variable values *low* and *high* in the following way: $U_{memory}(conf) = A_{high} \times U_{mem\ high}(conf) + A_{low} \times U_{mem\ low}(conf)$, where A_{high} and A_{low} are integer values, and $A_{high} + A_{low} = 1$. That is, when the reasoning variable has the value *low*, $A_{low} = 1$ and $A_{high} = 0$. This cancels the possible contribution of the configuration when the variable value is *high* and leaves the contribution only when the variable value is *low*.

The utility function I propose here for Cordelia's ARCM is rather simplistic, but clearly illustrates the intention behind utility function based reasoning. Real utility functions may involve several parameters interacting at the same time, and a combination of continuous and discrete functions. Reasoning over historic events is possible using utility functions, however, the account of memory requires defining complicated schemes on how to dynamically change the utility mapping.

Sometimes, the utility function cannot be expressed as a single function but as many functions [45]. In this case, the best configuration is the one that maximizes the value of each function. However, the optimization of one function can imply the minimization of other, thus it is necessary to make a compromise. This problem has been studied by multi-objective optimization [202], whose goal is to reach the best compromise among the different function values.

Utility function based reasoning is based on a predefined mapping between the possible configurations and the possible environment variations. This mapping is manifested as a utility or optimization function that is capable to discriminate between several configurations, which one is the best. The main benefit from utility function reasoning comes from the fact that it evaluates existing configurations without constructing them. Therefore it can be used to quickly pick one out of many pre-made system configurations. This is also the main drawback of this reasoning technique. In order to construct the utility function it is necessary to explicit a quantitative value for each configuration element and combination of element for each possible (or given) set of environmental conditions.

Other approaches

Researchers have explored a variety of techniques to reason in *self*-adaptive system. Recent work centers on using test at runtime [107], machine learning [128], and stochastic statistic [35, 75] approach to drive the system configuration change.

In [107], researchers explore the use of online testing technique to harness what they call *proactive self-adaptation*. Online testing means executing test during the execution of a *self*-adaptive application. Whenever a test fails, it points to a potential problem and then the system can proactively trigger a series of reconfiguration meant to prevent the system to fails in real execution. The underlying idea is to detect changes and deviations before they can lead to undesired consequences.

Recently, researchers have explored the use of stochastic approaches to drive the adaptation process [35, 75]. More precisely, one approach [35] uses Discrete Markov Chains [148] (DMC), a discrete random process with the following property. The probability distribution for the system at the next step (and in fact at all future steps) only depends on the current state of the system, and not additionally on the state of the system at previous steps. DMC may predict the probable future state of the system (statistical properties). The underlying idea is to construct a model (DMC) of the system, dynamically adjust the system parameters of this model to align it with its state, environment and objectives, and use it to make decisions that consider the probable future.

Similarly, other approach uses mathematical models that deal with non-functional properties, such as reliability and performance [75]. It consists in keeping models (Discrete Time Markov Chain – DTMC) alive at runtime and feeding a Bayesian estimator with data collected from the running system, which update the DTMC parameters. Updated models (at runtime) provide an increasingly better representation of the system that allow to detect or predict if a desired property is, or will be, violated by the running system. Property violations may trigger automatic reconfigurations (recovery actions aimed at guaranteeing the desired goals).

Discussion

Rule based, goal based, and utility function based reasoning are three wide spread reasoning techniques for *self*-adaptive systems. Each of these has their benefits and drawbacks. Rule based reasoning uses a situational strategy, in which predefined responses are triggered by predefined events. Yet, rule based reasoning can also consider historic events and through meta-rules evolve according to the past. Unfortunately, it is hard to foresee how rules are going to interact, especially when hundreds of rules are responsible for making a decision. In contrast, goal based and utility based reasoning use very different strategies. The first one pushes forward the concept of goals that drive the reasoning process, and uses these goals to derive each particular decision. It leaves situations thinking for the sake of accomplishing a set of goals. The second one defines an a priori

mapping between reasoning variables and configurations. It seems similar to situational reasoning, however, it is different because it does not derive but evaluates a decision (it picks one out of many options). Furthermore, it can select the best configuration given the encoding in utility function, rule based and goal based reasoning cannot ensure that. Finally, new reasoning techniques appear in the horizon. These techniques use stochastic models and online-testing to drive the system adaptations and produce promising results. Nevertheless, they are still immature and subject of current ongoing research.

In this thesis, I set rule based reasoning as the default reasoning techniques. The rationale that motivates this decision is that rule based reasoning properly illustrates the relationship between particular environmental conditions, historic events, and decision making in the reasoning process.

2.1.3 Adaptation Mechanism

> *– Cordelia is working at a customer's office, she uses her laptop computer to process requirement and schedule upcoming meetings. Suddenly the computer losses the wifi connection and engages the 3G network. The outlook calendar client closes and the Googlecalendar client opens up displaying the same displayed by outlook. A pop-up displays the following message "You are now connected to an insecure network, the ssl encryption protocol is now active".*

The previous excerpt illustrates how Cordelia's ACRM system adapts itself to a varying environment. When the wifi connection went off, the system engaged the 3G network. The system's reasoning engine then decided based on the service availability, security level, connection speed, etc. that the *outlook client* was not fitted for the new environmental conditions and the *Googleclient* should be opened instead. I have explained and surveyed the operation and techniques of reasoning in *self*-adaptive systems to that point. Now I address the final part, the *adaptation mechanism*.

A *self*-adaptive system's adaptation mechanism is the part that controls how the reconfigurations (or modifications, adaptations) are actually performed. In the case of the previous excerpt, the adaptation mechanism is responsible of: closing the *outlook client*, replacing the outlook connector with the Googleconnector, opening the Googleclient, activate the *ssl encryption protocol*, loading the previous data, and displaying the pop-up message. Notice that the reasoning engine makes a decision and the adaptation mechanism executes that decision.

In general, the adaptation mechanism manipulates the reconfigurable or adaptable parts of the system. In the case of Cordelia's ACRM system, the adaptable parts are the calendar client, the encryption protocol, and the messaging system. Each of these variable parts can be replaced with several options, some options can be incompatible with other options, or imply that a particular option should be adopted. Table 2.3 illustrates the different options for each variable part of the system. Notice that the mapping between

actions (decisions) and actual reconfigurations is not a one to one mapping. For example, the adaptation mechanism reflects the decision to change a particular calendar client by changing not only the client interface, but also the interface that allows the client to communicate with a particular server. The decision to use the *outlook* client will imply the use of the exchange server connector. One decision may imply plugging and unplugging several pieces of software.

Whenever the environment changes, the reasoning engines decides to adopt particular system options (cf. Table 2.3), and then the adaptation mechanism effectively puts in place these options. In this section, I survey the various mechanisms that have been proposed to realize the adaptation mechanism. Particularly I focus on two mechanisms: *adaptation through aspect-oriented programming* and *model driven adaptation*.

Reflection and Software components

In the early days of *self*-adaptive systems, reconfiguration and change was played using hard coded ad hoc adaptation mechanisms. These mechanisms worked by selecting a predefined and fixed number of configurations or systems modes, and switching from one mode to another by changing some parameters [27, 28, 97].

When computational reflection was introduced as a way to describe the overall architecture, state, and operations of the system [89, 147], researchers started to replace the old hardcoded adaptation mechanisms. A recent study shows that nowadays reflections conform the foundations of *self*-adaptive systems [7]. The main benefit of computational reflection is that it allows querying, and dynamically loading and unloading the different elements in software architecture.

Another improvement came from the introduction of component technology [215], which proposed the idea of configurable software systems by adding, removing or replacing their constituent components (sometimes using reflection). There exist several component models that support reflective capabilities, dynamic loading and unloading of various components. Among these component models we find FRACTAL [31], OpenCOM [60], OSGI [177], EJB [33], and so on. Basically a component is composed of ports, which require and propose services via particular interfaces. A component functions as a unit

Table 2.3: Cordelia's ACRM variable parts and their options.

Variable Part	**Options**
Calendar Client (and connector)	*ical*, *oracle*, *entourage*, *google*, *palm*, *outlook* *a4desk*, *groupwise full*, *groupwise lite*, *sunbird*
Encryption Protocol	*ssl*, *tls*, *kerberos*, *ike2*
Messaging System	*sms*, *voice*, *pop-up*
Server Connection	*exchange*, *google*, *ical*, *oracle*, *groupwise*

responsible of a particular computation. Then, a system can be described as a collection of connected components that deliver a service.

Reflection and component-based architectures widely improved the construction of adaptation mechanisms to achieve dynamic reconfiguration [6, 140, 59, 57, 66, 9]. These mechanisms use component models to control and visualize the overall architecture. The components in the component model and the actually executing system are causally connected, and changes in one affect the other.

Despite the introduction of components and reflection, the constructing of software capable of changing at runtime continued to be challenging. Performing adaptations required writing complex program that made use of the reflective operations needed to reconfigure the system. Therefore, reconfiguration becomes rapidly an ad hoc program that used the means of a reflective reconfigurable platform. Researchers addressed this challenge by proposing several languages that allowed manipulating high-level architectural elements instead of reflective operations. Among these languages we find the *Plastik* language [11] build on top of OpenCOM, and the *FScript* language [65] built on top of Fractal.

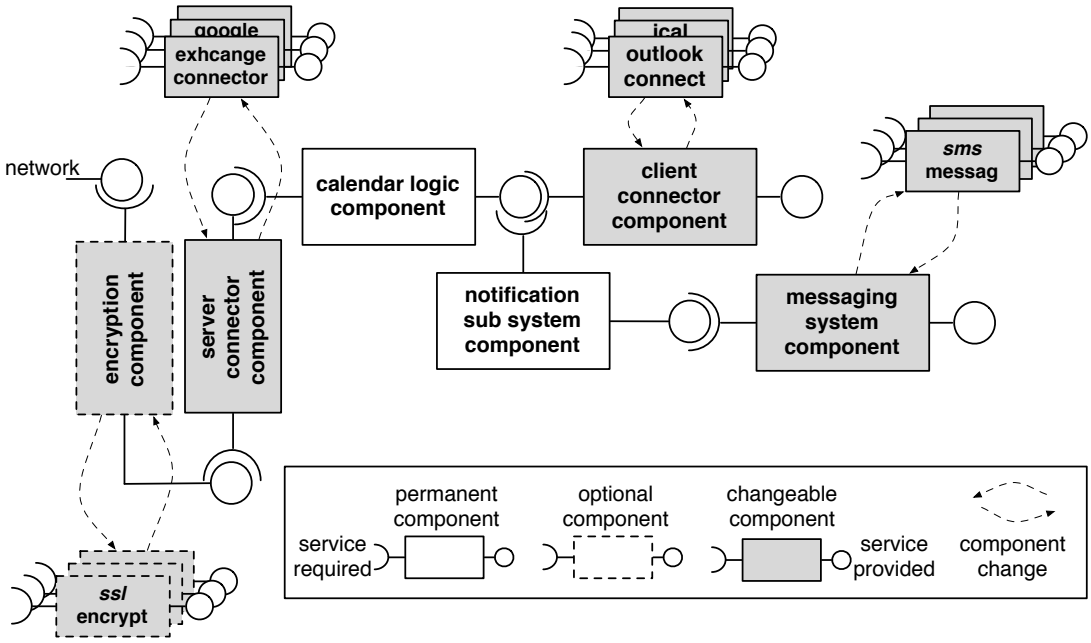

Figure 2.3: Component diagram of Cordelia's ACRM system.

Figure 2.3 illustrates the components of Cordelia's ACRM system. In the diagram, six components compose Cordelia's ACRM system. Out of them, four (gray boxes) can be replaced by other components offering and requiring the same services (in the figure connected with dotted lines), but with small differences. For example, the encryption component can be replaced by a component realizing the *ssl* or *ike2* encryption protocol. This

implies that multiple versions of a component can be used according to the needed functionalities. The component *client connector* is available in ten different flavors corresponding to the ten calendar client options (cf. Table 2.3). The component *server connector* is available in five versions corresponding to the different calendar services that can be available. In summary, when the reasoning engine decides to change the *outlook* client will the *google* client the adaptation mechanism swaps the *outlook client* connector component by the *Googleconnector* component. Besides it changes the server connector accordingly. One decision implies many changes.

Aspect-oriented Programming

Aspect-oriented programming [126] (AOP) was raised in the late 90's as a solution to improve the modularization of crosscutting concerns. Several languages and solutions have been proposed to implement AOP, each of these provides different constructs [109, 8, 25]. In general, two constructs are common among the different languages: *point-cuts* and *advices*. Point-cuts are predicates that designate the places where the crosscutting concerns are located, whereas advices are the realization of the crosscutting concerns.

AOP arises as an option, or even a complement to component platforms and reflection. A survey of adaptation of middleware platform using different compositional approaches such as aspects is presented in [151].

The composition mechanisms that allow aspects to be woven into the program to introduce crosscutting concerns have been used as a reconfiguration mechanism for adaptation [182, 64, 211, 220, 180, 191]. AOP has even been ultimately used to weave and unweave adaptation concerns leaving components implement the business concern [64, 198]. The underlying idea is that each aspect corresponds to a changeable dimension of the system, then these aspects can be woven / unwoven at runtime when required [184, 76, 210].

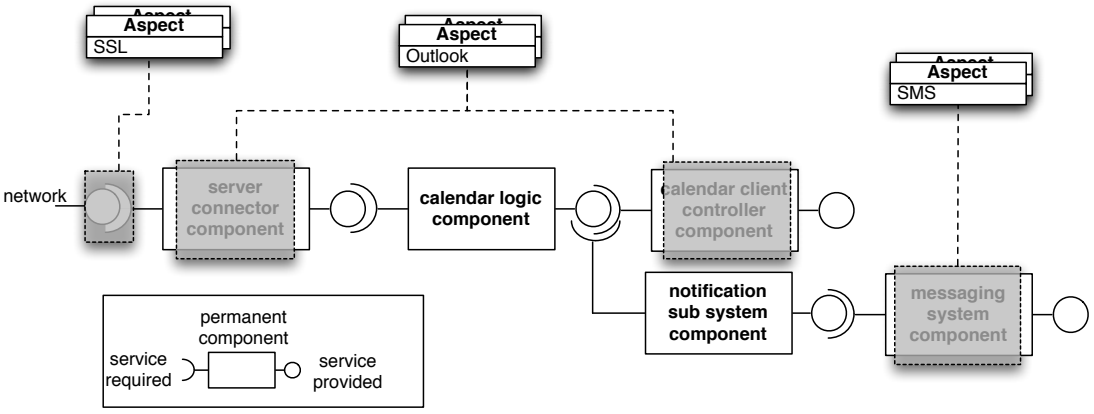

Figure 2.4: aspect-oriented adaptation of Cordelia's ACRM system.

Figure 2.4 illustrates how aspects can be used to adapt Cordelia's ACRM system. The figure presents a component diagram similar to Figure 2.3, however, this time there are no changeable components. Instead, several aspects (gray translucent boxes) may be woven / unwoven in order to reconfigure the system. This is possible because aspects' advices can replace or augment the behavior of a particular component (base program). These base components are indicated by point-cuts (semi-transparent gray boxes), which state the precise place and moment where to weave the advice. For example, one aspect's advice realizes the *outlook connector* behavior, whereas others may realize the *google*, *ical*, etc. connector behavior. When woven, these aspects insert their behavior into the *calendar client connector* (and *server connector*) component (as pointed by the point-cut), fitting the component with the intended behavior. A particular case happens with *encryption protocol* component, which has disappeared. Instead, an aspect's advice realizes the encryption protocol and when it is woven, it captures the data flow from the server connector to the network and encrypts the data. Then, when the reasoning engine decides to change the *outlook* client for *google* client, the adaptation mechanism unweaves the *outlook client* aspect and weaves the *Googleconnector* aspect.

AOP is well suited to realize the *self*-adaptive system's adaptation mechanism. Nevertheless, three major problems prevent the adoption of AOP as an adaptation mechanism. The first problem, known as the *AOSD evolution paradox* [217, 132, 150], occurs when aspects and the base system evolve separately. It consists in aspects advising undesired system's modules or not advising the desired system's modules. This problem originates in the insufficiency of current point-cut languages to abstract over structural properties of the base program. For example, if Cordelia's ACRM evolves by changing structural specifications of its *calendar client connector* component, the *outlook connector* aspect will miss the component and will not adapt the system properly. The second, known as *aspect interference* [121], happens when several advices are woven into same system point, or when an advice cancels the effect of other advices. For example, an advice that decrypts the communications can be woven after the SSL advice in the service connector port. Finally, the third relies on two properties of AOP: *obliviousness* [80] (the ability of aspects to interject behavior without being asked for) and *invasiveness* [164] (or the ability of aspects to break the object-oriented encapsulation). Invasiveness is a powerful property that enables AOP as an adaptation mechanism because it allows replacing and augmenting the components' behavior. However, when used obliviously it can also introduce undesired behavior without developer knowing about it.

Later works proposed another view of aspects, which reduces the previous problems. Instead of implementing reconfiguration as aspects, aspect's advice are a set of reflective operation that need to be performed to change the system [19].

Model Driven Adaptation

Model driven approaches to adaptation mechanisms focus on using models, or abstractions closer to a particular domain than computing (or hardcoded). These adaptation mechanisms are founded in Model Driven Engineering (MDE) [203] and use a variety of modeling facilities to describe the *self*-adaptive system's elements, adaptation, and relations between elements. The underlying idea is to abstract the adaptation mechanism from ad hoc reconfiguration scripts, reflective component management, and reconfiguration modes selection.

Early work in model driven adaptation mechanisms proposed using architectural models to monitor and control the system evolution [176, 175, 70]. Rainbow [90], an architecture driven approach provides reusable infrastructure together with mechanisms for specializing that infrastructure to the needs of specific systems. Adaptation is hardcoded using adaptation operators (set of specific actions that control the architecture), and adaptation strategies (constrained by operators and properties). This initial work applied at design time, whereas the code facilities to manage the system at runtime where semi-automatically generated.

More recent work keeps the design models alive at runtime [16] and uses them to drive the adaptation. In this way, models used at design time can also be used at runtime (the running system and models coexist) as artifacts to perform and control architecture-based adaptations [92, 222]. Besides, models can be used to verify and simulate reconfigurations at runtime.

A model particularly used to model *self*-adaptive system's variable parts comes from software product lines (SPL) [48]. As we mentioned in Section 2.1.1, SPLs are typically used to model (in a variability model) the various varying part of a software product [48]. These varying parts are called *variation points*, and represent the different system modules, components, or elements that may change from one product to another in the same family. Selecting specific options among the different variation points then derives products. Using software product lines to model the *self*-adaptive systems' underlying structure is appropriate since *self*-adaptive systems are also composed of changeable parts [15, 181, 81, 36].

The principal use of SPLs in adaptive systems is to represent the system variability at design and runtime [15, 17, 21]. At design time, variability models can be used to foresee conflicts, desirable, and undesirable configurations [18, 102, 83]. However, the real utility of variability models comes at runtime, where dynamic software product lines are used to derive a new product on the fly [101]. A dynamic software product line is basically a design time SPL that is kept during runtime, and then when the system needs to change (and a set of variants are selected), reconfigurations are computed automatically from the variability model resulting in a new system configuration (product) [37]. The changes in this new product are then translated into an architectural model (variability models assist the execution strategy to determine the step that are necessary to reconfigure the software system), and then reconfiguration scripts are automatically generated and the

system reconfigures [39, 38].

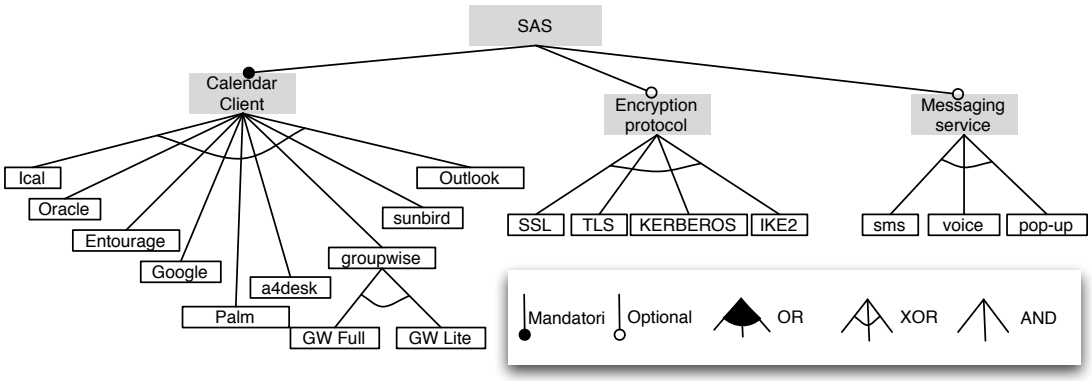

Figure 2.5: Product line variability model for Cordelia's ACRM.

Figure 2.5, illustrates Cordelia's ACRM system software product line. Notice that the variation points and options correspond to the ones introduced in Table 2.3. When the environment changes, the reasoning engine decides how to adapt the system. This is, it selects particular options on each variation point. This selection is then used to derive a product and reconfigure the system accordingly. For example, when the reasoning engine decides to change the *outlook* client for the *google* client, it selects the options *google*, *ssl*, and *pop-up* in the variation points *calendar client*, *encryption protocol*, and *messaging system*. The system derives a new system configuration that contains these options. Then, the system automatically generates the reconfiguration scripts needed to reconfigure the running platform.

Using SPLs at runtime and generating reconfiguration scripts allows producing unanticipated variability and interdependency relationship between variable artifacts. It poses certain problems at the moment of generating the reconfiguration scripts, since there is not necessarily a one to one mapping from the system architecture and the software product line that represent the system. This translation is not always straightforward and can impose some limitations on the derived products.

A more recent proposition consists in using models at runtime, model composition (similar to aspect-oriented mechanism), and causal links as adaptation mechanism [158, 155, 157, 156]. Before describing this novel model driven adaptation mechanism, I will describe its bricks *models at runtime*, *model composition*, and *causal link*.

Models at runtime, as we mentioned previously, consists in keeping any model alive at runtime (while the system is running) [16]. For example, architectural component models and software product lines are kept alive while the system is running. Other models, such as behavioral or stochastic descriptions of the system can also be used at runtime.

Model composition is an approach conceptually similar to aspect-oriented programming. It is intended to reinforce the separation of concerns by encapsulating different

views of the system in different models [117]. These views correspond for example to crosscutting concerns such as security and persistence, to other more precise views such as encryption. The fundamental idea is to separate the main business concerns among themselves, and from other non-business concerns. In particular we may see that this adaptation mechanism uses aspects models as sets of architectural modifications that realize a variant concern (or view) [19]. So far, many approaches have been proposed to compose models of different nature, language, and size [87, 171, 47, 193, 194, 159]. Composition can be *symmetric* or *asymmetric* depending the nature of the models to be composed. When the composed models conform to the same meta-model[†], then I refer to a symmetric model composition, otherwise, the composition is asymmetric and one model introduces elements into the other.

Causal link is a concept that consists in keeping the runtime models and the running system connected by a causality network (or series of events). The underlying goal is to keep the runtime model up to date and synchronized with the running system. In the vertical downside, causal link generates the necessary reconfiguration scripts or actions needed to update the running system. In the upside, causal link refreshes the model elements whenever is necessary. Several works offer solutions to keep a consistent model representation of the system, and propagating changes when needed using causally-connected models [185, 221, 207, 174, 157, 156].

In a general overview, this adaptation mechanism uses models at runtime to describe the system and the different varying concerns. Model composition transfers a product derived at runtime to the actual component model that reflects the running system. Notice that for this particular model composition, the models to be woven in are called *aspect models*. Aspect models here are similar to the ones described in [19], that is, they are sets of architectural-level reconfigurations. Causal link is used to keep the model at runtime up to date and synchronized with the running platform. This kind of adaptation mechanism helps developers and engineers to better coping with the complexity involved in constructing adaptive systems. They allow engineers to separate the different concerns in abstract units (models) and handle the problem of the combinatorial explosion produced by the large number of configurations that the system can adopt.

Figure 2.6, illustrates the aspect-oriented model driven adaptation mechanism. Initially, a component model of the system is kept alive at runtime (A). This component model reflects the running system state and configuration, as well as some platform specificities. A variability model at runtime describes the different system's variation points (B). This model is later used at runtime to derive a new product on the fly. Each variation point option is associated with one or more aspect models (C). When variation points' options are selected, the system derives a new product, which drives the weaving of the aspect models related to the new configuration (D). This results in a new component model, which is out of date with the running system (E). Finally, the initial and final component models are compared and their differentials calculated [144, 131, 228] to generate a set of reconfiguration scripts. These scripts are used to reconfigure the running platform and harmonize

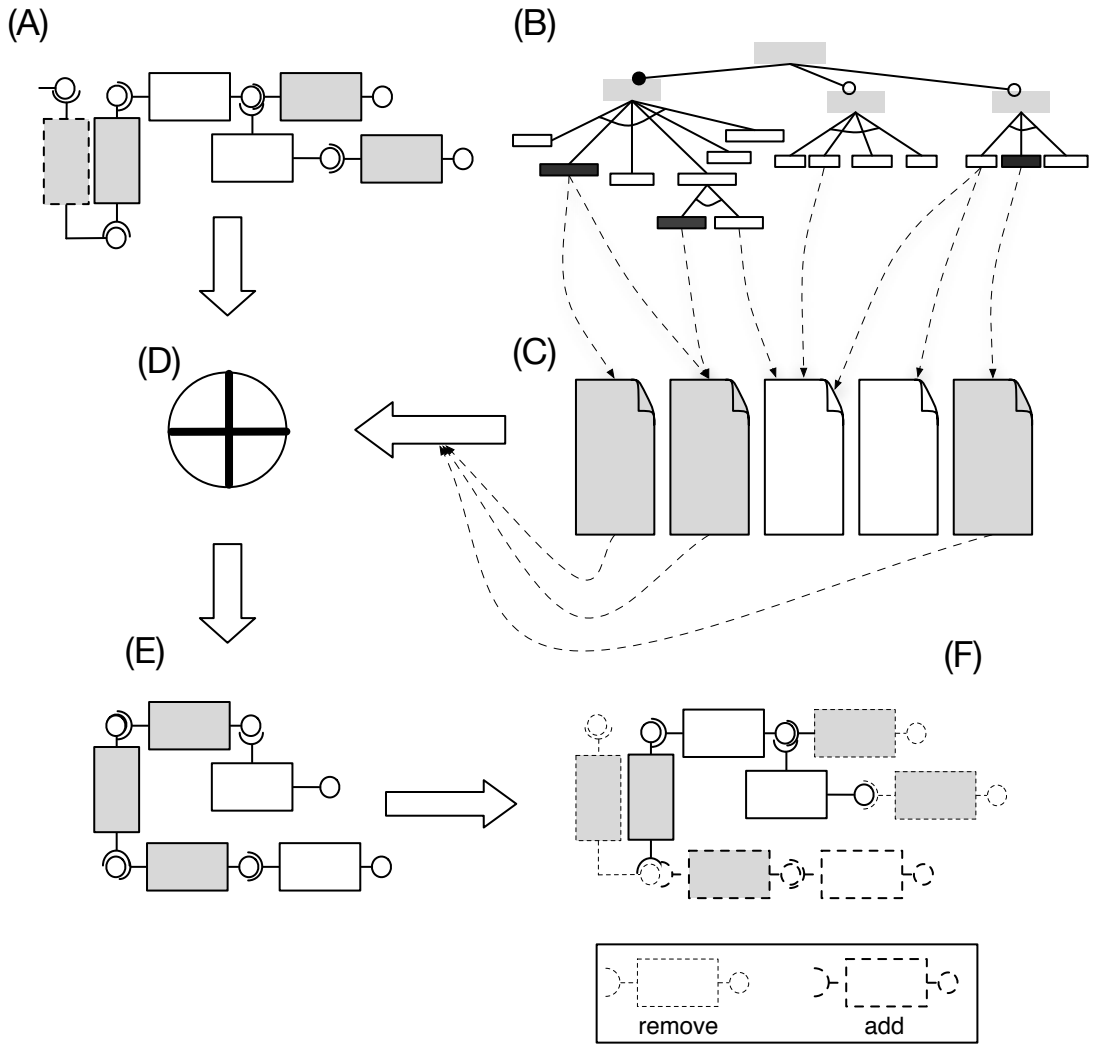

Figure 2.6: aspect-oriented Model Driven adaptation mechanism

the running system with its model (F).

2.2 Validation and Verification of *self*-adaptive system

In the previous section, I have illustrated the different parts and techniques to construct *self*-adaptive system. Each of these parts is critical for the well functioning of *self*-adaptive systems; hence, ensuring that they work correctly is extremely important. Reasoning engines must make the right decision in all environmental conditions, and adaptation mech-

anisms must correctly apply the adaptations that the reasoning engine dictates. Faulty adaptations would yield an unadapted system that is unsuited for its purpose, whereas faulty decisions will leak into faulty adaptations. Notice that I assume that the environmental data is pristine and contains no fault at all. This is a strong assumption that allows me to focus and analyze the reasoning and adaptation processes.

Reasoning engines and adaptation mechanisms can be validated and verified in order to ensure their correct function. Yet, validation and verification are different concepts that serve the same purpose. According to the IEEE [93], verification is the confirmation by examination and provision of objective evidence that specified requirements have been fulfilled. That is, verification addresses the correct functioning of a software system with respect to a requirement specification, stated in several properties. Validation on the other hand is the confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled. That is, validation addresses the correct functioning of a software system with respect with a particular use requirement. Validation is typically conducted through the execution of test cases that reflect the requirements for particular system uses. In this thesis, I focus on the validation through testing of reasoning engines and the validation through specifications of aspect-oriented adaptation mechanisms.

In this section, I present the state of the art on testing and verification techniques for adaptive system. The different techniques that I present here apply to the validation and verification of *self*-adaptive system directly or indirectly. Most of these techniques address the subjects of validating or verifying *autonomous reasoning* and *adaptation mechanisms*. I also present the state of the art solutions to solve the problems that affect aspect-oriented programming and that hamper its adoption as an adaptation mechanism.

2.2.1 Testing

The testing activity is one of many ways to validate a piece of software [23]. It consists in empirically assessing that the realization of a software system works as it is expected to [183]. Such expected behavior as well as the stimuli that trigger that behavior must be known a priori by testers. Figure 2.7, illustrates the testing activity. It goes as follows: the tester (A) feeds the system (test subject, or the subject of the test inquiry) with testing data (or input data) such as *integer number*, *string data*, *models*, etc. He/She (B) waits until the subject execution is completed, and then (C) evaluates the resulting observable behavior (it can be the computation result). The sum of these three steps is referred as a test cases, a triple composed of *test data*, *execution conditions*, and *expected behavior* (or result). Since execution conditions are typically the same for all test cases, they are omitted in the test case definition.

Some steps in the testing chain can be automated. For example, data can be generated automatically [133], complex execution schemas can help to feed input data and collecting results, and finally, an automated *oracle* can help testers to evaluate the results. Yet,

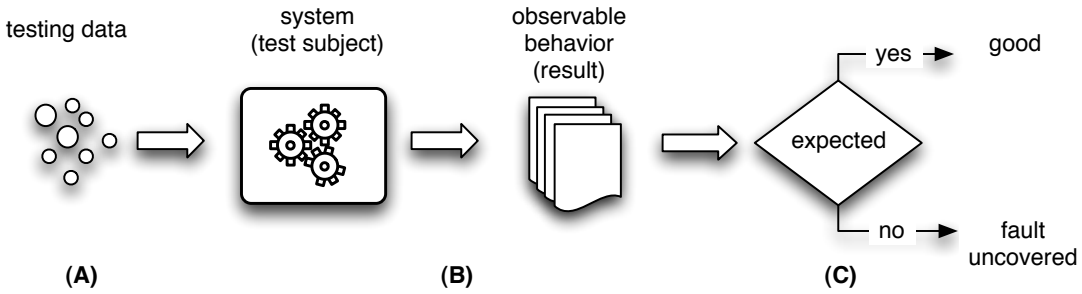

Figure 2.7: Testing practice parts.

automating all these steps results being challenging [22].

There exist two kinds of testing techniques, those that consider the testing subject as a black-box, and those that consider the testing subject as a white-box. The first kind is applicable to almost any system without particular interest in its implementation details (the system is a black-box). The second kind considers the implementation and structural details for constructing the test cases (the system is a white-box [3]). Most approaches on testing *self*-adaptive system deal with the automatic generation or synthesis of data by using either black-box or white-box criteria. In the following I present and survey the state of the art on testing *self*-adaptive system.

Testing on decisions and paths

Automatically generating test input data for testing *self*-adaptive system is possible because there exist *coverage criteria* [23]. A coverage criterion is a set of properties that a data set must satisfy in order to ensure some degree of coverage with respect to structure or domain.

Structural coverage criteria are based on the program's specific structure to determine whether a data set is good [4] or not. Several criteria are based on the different path data or execution can follow in the program, one of these criteria is the All Path criterion [14]. It requires that data trigger every possible execution path in the program. In imperative programming it implies covering decisions, statements, and entry/exit operations. In rule based reasoning, it implies covering the possible decision outcomes and reasoning paths (chains of reasoning rules).

For example, consider rules R1 and R2 (listing 2.1 and 2.3) from Cordelia's ACRM system reasoning engine. The number of paths these rules can follow is small. Either R1 is executed and not R2 (P1), R2 is executed and not R1 (P2), or both are executed R1 first

3. White means that the system structure is transparent and visible

4. Good in this terms refer to the ability of the data set to ensure that fault will be revealed, in this case dictated by coverage of a particular criterion.

then R2 (P3), or R2 first and then R1 (P4). Since there are no shared variables it is possible to execute both rules consecutively, otherwise the execution would have two variables. A data set satisfying the All Path criterion should contain data that may trigger the reasoning paths P1, P2, P3, and P4. Notice that R2 is a rule with conditions over temporal events and requires to be satisfied by at least two environment instances.

Researchers have extended the traditional path coverage criteria to formulate constructions adapted to rule based systems [99, 4, 129, 10, 100].

Early approaches on path coverage proposed selecting sets of test data that exercise the structural components of the rule-base as exhaustively as possible. This involves firing all rules, and also firing every causal sequence of rules. Path hunter uses structural path analysis to detect potential interactions between rules in a rule-base and to identify problems within the rule-base, essentially using path enumeration [99, 4].

The logical path graph model [129] is a rule's execution path model based on control flow analysis. It attempts to determine a set of paths through the rules such that when executed, they would adequately test the rules and their interactions. These paths are then used to generate testing data that cover the rule execution and interactions.

Path coverage also spans to the causal relation and interference between rules. Such extension defines five coverage measures that deal with the rule conditions and actions. It evaluates the coverage of rule execution paths, of rules causal relation, and rules interference. These coverage measures are used to assist rule pruning and identification of class dependencies in addition to guide the automatic generation of testing data [10]. Such testing data should cover all rule's execution paths, their relations, and possible interferences (and resolutions). A tool chain for generating, executing, and evaluating test cases for rule-based systems is proposed in [100].

The continuous evolution of *self*-adaptive systems imposes some difficulties to existing software system test. Existing test suites may not be appropriate for the new system evolution; they can contain superfluous or missing test cases that may render the testing hard. A possible solution is defining a coverage criterion by identifying the paths of change propagation [201]. This will allow tester to handle multiple and complex changes to identifying new or modified behaviors, which are not exercised by existing test.

Other researchers have studied the synthesis of test data for context-aware software systems as a whole. This synthesis uses a model of how the program handles contexts that are continually checked and resolved by context inconsistency resolution (context-aware application may be affected by buggy context data). Based on this model, a context-aware data flow analysis proposes a set of equations to analyze the potential impact of context changes and serve as test adequacy criteria for testing data [106, 145].

Test suites in a context-aware application can be improved by identifying the points in the system that are affected by context aware data by systematically manipulating the context data fed into the system. The application of these manipulations increases the system exposure to potential valuable context variations, which may lead to better test cases [223].

Testing by search

Search-based software testing (SBT) is the application of search-based software engineering (SBSE) [103] to address testing issues. Basically, SBSE consists in mapping an optimization function to a particular software engineering problem. Then, use search-based meta-heuristics such as genetic algorithms [96], simulated annealing, etc. to find the better value for that function, which corresponds to the best solution for the problem in question.

SBT has been applied to a wide variety of issues, including testing of structural [206, 179, 152, 79, 118, 153, 179], functional [224, 152] and non-functional properties [225, 152]; interaction [51, 55, 119], mutation [12], and regression testing [142]; and recently agent systems [172]. Of particular interest for *self*-adaptive system are structural, functional, and interaction testing using search techniques.

Using search-based techniques to solve testing problems provides several benefits such as *fast solving time*, better coverage for functional and non-functional testing, etc. Yet, SBT has a few drawbacks: it is necessary to *encode testing problems into an optimization function* and *adapt meta-heuristic search technique to the particular problem slang*†. The different flavors SBT can adopt are equivalent to the different ways to map a problem into an optimization function.

For example, search-based structural testing consists is generating testing data by optimizing a function based on the program structure. Such function builds upon a quantification of program structural elements, such as if statements, loops, assignments, etc. in relation with data. Therefore, meta-heuristics search the space (that contain all the possible program inputs) looking for the best set of data that provides the best value for the function quantification.

A particular case, which is suited for testing rule-based reasoning uses data flow graphs to quantify the program structure [152]. More precisely, the flow graph is used to derive a set of constraints about the program variable values, which describe the different parts of the system that are triggered by a simple data value. The resulting optimization function counts the number of satisfied constraints and provides a quantitative measure of how good or bad a data set is. To illustrate this idea, consider rules R1 and R2. I skip the creation of a flow graph immediately to derive the constraints:

$$C1\text{: } platform = windows \;\&\; use\ type = reserve \;||\; platform = mac \;\&\; use\ type = reserve$$
$$C2\text{: } exchange\ service = available \;\&\; sec\ level = low \succ google\ service = available \;\&\; security\ level = high\ .$$

Notice that these constraints are defined in a temporal context. The symbol $\succ$ indicates that one value must follow the other in a temporal line. This implies that the resulting data set must contain at least two environment instances. The sum of the satisfaction of C1 and C2 constitutes the optimization function. Then, the search consists in finding a sequence of reasoning variable values that satisfy these constraints. The optimization function's value

directs the search by discriminating the worthiness of the between data sets. A data set that satisfies two constraints is better than one that satisfies only one.

SBT for black-box testing and functional testing focus on different aspects of the program. Black-box SBT searches to satisfy and optimize a particular coverage criterion, which is based on a description of program's input domain.

Reasoning is not only present in *self*-adaptive systems, autonomous agents are also capable of reasoning and making decisions. SBT has been used to test whether autonomous agents make the right decisions through test case optimization [172]. This consists in modeling the agent autonomic requirements (for example, the tasks the agent will perform in a given context) into a quality function. Then, a set of initial test cases is evolved using the quality function as a minimization function. The lower the values of the quality function, the stronger test cases are. The optimal search value is a set of test cases that test the system in a large variety of environments.

Testing combinatorial interactions

Structural testing techniques do well in handling test data generation when the program structure is known. When the program structure is unknown, or when the test subject underlying implementation can change, it is more convenient to use a black-box testing technique. Black-box testing relies on descriptions of the problem domain, input variables, constraints, and so on to generate data.

Combinatorial interaction testing (CIT) [136, 55, 119] is a series of back-box testing techniques used to sample testing data from large spaces. For example, Cordelia's ACRM system environment has 6.144 instances, which with just a few variables raises fairly sized space (notice that a system with 20 reasoning variables, with 5 possible values each will raise a space of $5^{20} = 95.367.431.640.625$ instances, which is very large). CIT consists in selecting combinations of variable values (interactions between variables) in such a way that all the combinations (interactions) are covered at some degree. For example, covering the two-combinations of variable values (degree of coverage is two) will require testing data to contain all the pair of variable values – this is often called *pairwise*. Since the interactions cited by most work on CIT occur between different variable values, I call them *inter-variable interactions*.

The underlying premise of CIT is that variable value conventions can uncover faults in software systems (this is, many faults are caused by combinations of variable values). A study shows that all the failures in a system could be triggered by a maximum of four to six-way interactions [136]. CIT has been used for testing several software systems [136], finding and characterizing faults in configurable software [233], testing applications with elaborate configuration options, such as web browsers and office tools [56], and testing GUI interfaces [231] among others.

There exist two prominent CIT techniques: *Covering Arrays* (CA) [50], and its extension *Mixed Level Covering Arrays* (MCA) [55], which embodies CA. The main characteristic of

MCA is that it supports the definition of variables with different domain size. CA only supports variables having the same domain size. This puts MCA in good way to sample a large data space such as Cordelia's ACRM system environment.

Adaptation mechanisms can also benefit of CIT. That is by testing that adaptation mechanisms are capable of producing configurations and that those configurations work. MCA can be used to sample data from a SPL, and select different sets of configurations. Still, this is not as simple as it appears. SPL may include several constraints that will make some combinations of option illegal. Additionally, variation point's options may contain dependencies and historical restrictions. Some of these inconveniences have been addressed by researches, particularly the modeling [52] and generation of covering arrays in presence of such constraints [53, 54, 94]. CIT has been applied successfully to test systems whose configuration evolves (changes) in time. More precisely, covering arrays have been used to select regression test for each new version of the system [188].

CIT has proven effectiveness and value on several applications, particularly pairwise. Several approaches have been proposed to generate CAs and MCAs [50, 34, 51, 55, 32, 119]; some of them are based on mathematical foundations and complex algorithmic, whereas others are based on heuristic search [51, 55, 119]. Nowadays there exist proficient tools to generate and use pairwise [34], but in order to obtain more confidence it is necessary to go beyond the pairwise [137] and cover the interaction of three or more variables. One of the main problems of increasing the degree of coverage is the amount of resources needed to efficiently generate covering arrays. A recent approach proposes to increment the degree of interaction coverage according to available resources [86].

2.2.2 Verification

Verification [93] is the confirmation by examination and provision of objective evidence that specified requirements have been fulfilled. Verification techniques for *self*-adaptive systems are deeply rooted in formal mathematical foundations and consist in checking that system conforms a set of properties or specifications [62, 138, 13, 68, 237, 238, 200, 212, 173, 95, 26, 229, 230]. Such foundations emphasize in representing a system as abstract and sound mathematical models, and performing exhaustive checking over such model.

With a mathematical model of the system in hand, engineers can check the system realization against the model properties (for example, temporal logic properties) [71, 2]. If the system realization satisfies these properties, then it is said to conform to its model and hence it is valid. Verification for *self*-adaptive system has two main axes: (1) verification of dynamic change, or whether the system conforms after changing, and (2) verification of reasoning, or whether the reasoning satisfies some soundness, completeness, and coherence properties. In the following I present the state of the art on verification for *self*-adaptive system.

Verifying dynamic change

Initial approximations in verification of change started with the verification of dynamic software architectures. The formal model of the system consists on a behavior specification associated with the description of components in a software architecture. Labeled transition systems specify behaviors and composition reachability analysis to check structural changes (before, during, and after the change). Then, an automated analysis verifies that changes in the system component do not violate the integrity of the model properties [135]. Another way to look at the problem consists in modeling the systems and its adaptations using lattices (partial order sets in which properties are specified) [62]. This lattice model is then used to verify whether the system satisfies lattice-invariant properties after adaptation [138]. Another possible model of the system consists in a graph of elements. Changes in the architecture are then represented as graph transformation that should preserve the safety structural properties and inductive variants [13].

Components are fundamental parts of *self*-adaptive systems. Changing one component by another or composing two or more components can produce several unforeseen behavioral variations. Researchers have proposed a verification theory for component composition. This theory aims at predicting that complex assemblies of component will hold their intended behavior [68].

Researchers have explored the modeling of change in *self*-adaptive system by using Petri nets for behavioral specification [236]. Such model contains specifications of adaptive and non-adaptive behavior. In these models, engineers use linear temporal logic (LTL) to describe non-adaptive behavior, and an extension adaptive-LTL [237] to specify the adaptive part of adaptive systems. Then a *self*-adaptive system is represented by a network of interconnected state machines, which represents the system and its changes. Whenever the actual system changes, it should be model checked against the properties defined in the model. Yet, such model can be very large and model checking can take a long time. One way to address this issue is by exploring the use of different data structure representation to improve the performance of the verification (which is degraded by the explosion of states) [216]. A recent proposition to address this kind of verification consists in modularizing the model-checking activity. Modular model-checking techniques identify the parts of the system that have changed and only verify those parts, and assume previous verification as still valid for the new system [238].

Self-adaptive system can also be represented as a network of states (finite state model), in which each transition between states corresponds to an adaptation and each state a system configuration [200]. This enables typical finite state properties such as termination, aliveness, etc. to be verified. Additionally, a series of fault adaptation patterns, interference between adaptation, and inconsistencies can be detected by navigating every transition.

Verifying reasoning

Early work on verification proposes verifying that rules in a rule-base do not conflict, are redundant, or subsumpt other rules. The underlying goal is checking whether a rule base is complete and consistent by enumerating decision and circumstances [212, 173]. This work is limited to the identification of static problems for atomic rules and cannot identify problems that result along longer reasoning chain.

Another approach, KB-Reducer, proposes to use an implied network of rules to represent the rule-base. In this network, the rules relations are represented explicitly in the form of a reduced directed graph. The process of reduction involves calculating all possible logically independent and minimal sets of inputs under which the knowledge base will conclude each assertion. This approach has the advantage of checking a rule-base for inconsistency and redundancy over inference chains, not just pairs of rules [95].

A more recent approach proposes to verify multi-agent programs whose behavior is specified in a rule-base. Agent behavioral models are expressed in a temporal language *shallow*, which extends LTL with agent related modalities. Then, the effective behavior of agents is verified against its shallow specifications [26].

Another line of reasoning verification focuses on assessing whether the environmental data is consistent. The underlying motivation is that environmental changes that the system acquires could be obsolete, corrupted, or inaccurate. To address this problem, researchers have proposed a formal model and algorithms for incremental consistency checking of the environmental properties [229]. These are later implemented in a framework for performing dynamic context consistency management [230].

2.2.3 Positioning with respect to Testing and Verification

In the previous sections I have presented the state of the art on validation and verification of reasoning and dynamic change in *self*-adaptive systems. These techniques have a series of limitations that leave room for improvement.

Autonomous reasoning

I have identified a series of limitation in the state of the art techniques to test and verify the autonomous reasoning of self-adaptive systems. In order to illustrate the gaps left by these techniques, I compare their limitations and capabilities in five dimensions: (i) *structure coverage*, or the property of the techniques to fully cover the structure of reasoning engines; (ii) *reasoning space coverage*, or the property of the techniques to cover the reasoning space in spite of the reasoning engine structure; (iii) *generality*, or the applicability of the techniques to a variety of reasoning techniques such as rule based reasoning, goal based reasoning, etc; (iv) *non-enumeration*, or the capability of the techniques to avoid enumerating every possible decision and decision path; and, (v) *coverage of the temporal*

dimension of reasoning, or the capability of the techniques to address the temporality involved in self-adaptive systems. Table 2.4 summarizes the comparison of the different state of the art techniques.

Table 2.4: Capabilities and limitations comparison of test and verification techniques for autonomous reasoning.

	Structural Testing [99, 4, 129, 10, 100, 201]	CIT [136, 55, 119]	Rule Verification [212, 173, 95]
structure coverage	Yes	**No**	Yes
reasoning space coverage	**No**	Yes	**No**
generality	**No**	Yes	**No**
non-enumeration	**No**	Yes	**No**
temporal dimension of reasoning	Yes	**No**	Yes

Table 2.4 evidences the following limitations: (1) since structural testing techniques use precise structural information to generate testing data, they are incapable of covering the reasoning space. If engineers writing reasoning rules were oblivious about some reasoning space's regions, then structural testing would not be capable of generating data covering these regions. (2) Furthermore, consequence of such dependency on structural information, structural testing techniques are tightly coupled with a particular reasoning strategy (in this case, rule based reasoning). (3) Verification techniques suffer a similar limitation because they rely either on the reasoning engine structure or on a formal specification of the reasoning engine. (4) Some structural testing techniques and verification techniques require enumerating all the reasoning paths and possible decisions. This is not always possible because the number of decisions and paths to those decisions can be huge. For example, meta-rules that modify, add, and remove dynamically reasoning rules may produce a huge number of reasoning paths. (5) CIT covers the reasoning space, provides generality, and does not require enumerating all possible decisions. Yet, it cannot handle the temporal dimension of reasoning. Since reasoning engines make decisions based on the history, CIT provides no guarantee to uncover errors due to faulty reasoning on historic events.

In this thesis, I propose a testing technique that addresses the limitations of the current state of the art techniques. More precisely, I extend CIT with a set of properties that enables the coverage of the temporal dimension of reasoning.

Dynamic change

I have identified the following limitations in the state of the art techniques to test and verify dynamic change in self-adaptive systems:

– **CIT does not handle the changes of system states and fails to cover the transitions between system configurations**. Since self-adaptive systems continuously

mutate themselves, it is important to ensure that the changes between the different configurations will not break down the system.

- **Component composition verification [68] requires knowledge of the behavior of all the components in order to assemble all the feasible compositions.** Such behavioral description is not always available, and there can be a huge number of feasible component compositions.
- **Verification techniques [236, 237, 200, 68] require enumerating all possible system states, and the transitions between these states**. This is not always possible since the number of system states and transitions between these states can be huge.

In this thesis, I do not address these limitations since I'm not interesting in the validation and verifications of dynamic change. Instead I focus on identifying and addressing the limitations for aspect-oriented based adaptation mechanism.

2.2.4 Specifications for aspect-oriented adaptation

AOP and its composition mechanism provide flexibility and versatility[†] for realizing adaptation mechanism. Engineers can use advices and point-cuts to change, add, or remove the system's structure and behavior on the fly. Nevertheless, three major problems – *AOSD evolution paradox*, *aspect interference*, and the combination *obliviousness - invasiveness* – prevent the successful adoption of AOP as adaptation mechanism.

Furthermore, these problems negatively impact the adaptation mechanism validation process. Undesired side effects, interference between aspects, and uncontrolled invasiveness make it difficult to ensure that when the system changes, the functionalities are preserved. Therefore, in order to verify whether new advices may preserve the functionalities it is necessary to re-execute functional tests. Additionally, if the system exhibits deviations from the correct behavior, AOP's problems make it difficult to trace the source of these problems to particular aspects.

Researchers have explored different options to solve these problems. These options can be summarized in three groups: *guidelines*, *aspect characterization*, and *modular systems*.

The first, guidelines consists in advises for developers stating how to write base systems and aspects [150, 217, 127]. These guidelines aim at helping developers to reduce the coupling of aspects with the base system and at encouraging them to carefully select advised system points. Such guidelines are meant to tackle interaction and obliviousness problems.

Aspect characterization consists in classifying the aspects according to their interactions with the base system and with other aspects. Direct and indirect interactions occur between aspects and methods [196]. Direct interaction is when an advice interferes with the execution of a method, whereas indirect is when advices and methods read/write the same fields. A characterization of aspects classifies them among *Spectative*, *Regulatory* and *Invasive* aspects according to their invasiveness degree [122]. *Spectative* aspects observes

and do not interfere with the base system, *Regulatory* aspects modify the program behavior in particular cases, and *Invasive* add or replace behavior in the base system. Another classification, *Spectators* and *Assistants* [49], proposes to control interactions by specifying the aspects that can advise the base system. It classifies aspects among *Spectators* (non invasive advices) and *Assistants* (invasive advices). Then, the base system explicitly demands the assistance of assistant aspects (identified by their names). *Spectative* and *Spectators* aspects are less likely to interfere with each other. On the other hand, invasive aspects, particularly those interacting at the same system points are more likely to interfere with each other. A fully automated approach can discover conflicts among classes and aspects directly from Java bytecode. Such technique uses a rule engine for identifying possible conflicts among advices, methods, and fields. The possible conflicts are represented by means of rules that operate over a knowledge base obtained through static analysis of classes and aspects bytecode [77].

Modular systems propose to establish interfaces exposing parts of the system to aspects. *Open Modules* [5] is a system that focuses on the exposure of specific join-points. This approach hides all the system points visible to aspect, and then each module must declare the system points it will expose to be advised. *Open Modules* exposes system points without distinguishing the aspects advising them, this makes developers aware about aspects advising the system. A similar approach, *XPI* [98], proposes using interfaces that mediate between aspects and the base program. Such interfaces establish a set of design rules to implement aspects and the base system in such a way that the evolution is coordinated through the *XPI*. *Aspectual collaborations* [143], also proposes to establish an explicit bridge between aspects and modules. Then, aspects are forced to collaborate with the base system if they are going to advice its points. The approach proposed by integration contracts functions on the same bases as collaborations, explicit definition of interaction [139].

Additionally to these attempts to solve AOP's problems, researches have proposed to assist developers when using aspects in different versions of a system [208]. Through an analysis that compares the changes in the set of matched join-points for two different versions of a system, this approach reveals unexpected changes in the matching behavior of system points. This analysis serves to assist developers finding bugs introduced by broken point-cuts.

2.2.5 Positioning with respect to the specifications for aspects

I have identified a series of limitation in the state of the art techniques to specify aspect-oriented adaptation mechanisms. In order to illustrate the gaps left by these techniques, I compare their limitations and capabilities in five dimensions: (i) *obligation binding*, or the capability of the techniques to bind obligations that developers must follow; (ii) *analysis support*, or the capability of the techniques to provide information that can be useful for analysis; (iii) *invasiveness control*, or the capability of the techniques to provide the means to control the aspects' invasiveness; (iv) *interaction control*, or the capability of the

techniques to provide the means to control the interactions between aspects and the base system; and, (v) *evolution support*, or the capability of the techniques to provide support to developers in case of evolution. Table 2.5 summarizes the comparison of the different state of the art techniques.

Table 2.5: Capabilities and limitations comparison of specifications techniques for aspect-oriented adaptation mechanisms.

	guidelines	Characterization	Modular systems
	[150, 217, 127]	[196, 122, 49, 77]	[5, 98, 143, 139]
obligation binding	**No**	**No**	Yes
analysis support	**No**	Yes	**No**
invasiveness control	Yes	**No**	Yes
interaction control	**No**	Yes	Yes
evolution support	Yes	**No**	Yes

Table 2.5 evidences the following limitations: (1) Guidelines bind no obligations to developers, and therefore, cannot provide guarantees about the developers' behavior. Developers can still write invasive and interfering aspects that introduce side effects. Yet, if developers follow these guidelines, they should be able to support the evolution and control the aspects interactions. (2) Aspects characterizations provide support for analysis and study of aspects interactions, but do not provide tooling support or bind any responsibility to developers. These characterizations do not guarantee that aspects will not interfere or that they will have no undesired side effects. (3) Modular systems allow developers to specify open points in the system and control the invasiveness, the interactions, and support evolution. Nevertheless, this control of invasiveness and interactions is coarse grained. Either developers open system's points to all aspects, or the close them to all aspects. Modular systems do not allow the developers to specify the kind of aspect to which the system points are opened.

Obligation binding, analysis support, invasiveness and interactions control, and evolution support are properties that are desirables in order to control the interactions between aspects and the base program, overcome the AOSD evolution paradox, and make aspects perform as expected. The limitations of the current state of the art techniques not only hamper the adoption of AOP as an adaptation mechanism, but also negatively impact the validation of aspect-oriented programs. In this thesis I address these limitations and propose a specification framework that combines the best features of the aspects characterizations and modular system.

2.3 Contribution of this thesis

In the previous section I have presented the state of the art techniques for constructing, validating (through testing), and verifying *self*-adaptive systems. I have also show that testing and verification techniques have many limitations. Validation techniques for reasoning engines are either too specific to a reasoning engine and do not ensure coverage of the reasoning space, or do not address the temporal dimension of reasoning and are rapidly surpassed by the possible number of environments. Specifications techniques for aspect-oriented adaptation mechanism address one AOP limitation and make no compromise to address the others.

In this thesis I address the previously announced points. First, I propose *multi-dimensional covering arrays* for testing reasoning engines. To formulate this technique I reuse the concepts from combinatorial interaction testing. Second, I propose a *specification framework* to address the problems of invasiveness and interactions of aspects. This framework is based on a fine grained characterization of aspects and tooling support integrated with state of the practice technology.

Chapter 3

Testing Reasoning Engines

Reasoning engines realize the core reasoning process in *self*-adaptive systems. These pieces of software make decisions that affect how the system interacts with its environment and how it satisfies its requirements. They make decisions over a reasoning space, which represents a changing environment. Such decisions are supported by domain knowledge and influenced by the own history of the system. Reasoning engines perceive and use history as a mean to support the decision making process.

Since reasoning engines drive the *self*-adaptive system's behavior given a changing environment, their decisions must be carefully reviewed. Therefore, assessing the correctness of the decisions made by these engines is critical. A faulty (incorrect) decision may lead to erroneous behavior, unfeasible adaptations, or system degradation. The testing activity helps engineers to achieve such assessment. It consists in simulating environmental conditions and changes. Then, for each possible environmental condition and variation, engineers review the reasoning engine's decisions to determine whether they are correct or not. If they find the decisions to be incorrect, then, they would find the problem, solve it and re-review the reasoning engine's decisions. This is repeated until covering the whole reasoning space.

Nevertheless, simulating every possible environmental condition and interactions among conditions is most often impossible because their number grows exponentially with the number of properties that model the environment. Additionally, time needed for the reasoning engine to make a decision and for the engineer to review that decision put another barrier to exhaustive simulation. Instead of simulating every possible environmental condition, engineers may select some of them that represent the environment [167]. Structural testing techniques allow engineers to select environmental conditions according to the reasoning engine's structure; however, they are specific to a single reasoning technique (cf. Section 2.2.3). Black-box strategies based on environmental criteria seems more appropriate to select representative environmental data. *Mixed level covering array* [55] (MCA) is a black-box strategy that can be used to sample large spaces and that can be used for

environment sampling.

I argue that the reasoning space comprises two types of interactions capable of inducing faults. These interactions are *inter-variable interactions* and *intra-variable interactions*. MCA effectively addresses inter-variable interactions [136], however, it fails to handle intra-variable interactions – temporal dimension of reasoning (cf. Section 2.2.3).

In this chapter I address the MCA's limitations and propose an extension to handle intra-variable interactions. First, I characterize these interactions. Next, I show the limits of MCA for sampling the reasoning space, and why a new technique is required. Then, I present multi-dimensional covering arrays (MDCA), an extension of MCA that handles temporal aspects of reasoning. Finally, I introduce three techniques to constructs MDCAs and an experimental assessment of MDCA's effectiveness.

3.1 Inter-variable and Intra-variable Interactions

The environment representation comprises several reasoning variables whose values can change over time. An environment instance – a snapshot of the environment at a precise moment – comprises a precise value for each reasoning variable. Since the combinations of these values can change a decision outcome, I argue that they interact. Therefore, I refer to the collection of interacting values of different reasoning variables as *inter-variable interactions*. In general, there are as many interactions as reasoning variable values. The reasoning variables in Table 2.1 produce 6.144 interactions (equivalent to the number of possible environment instances) between 11 variables. Nonetheless, a few of them can actually trigger decisions in the reasoning process. For example, consider the values *windows* and *reserve* for the reasoning variables *platform* and *usetype* in Table 2.1. The interaction of these variables triggers a decision in rule R1 (cf. Listing 2.1); however, the interaction between the values *palm* and *view* triggers no decision.

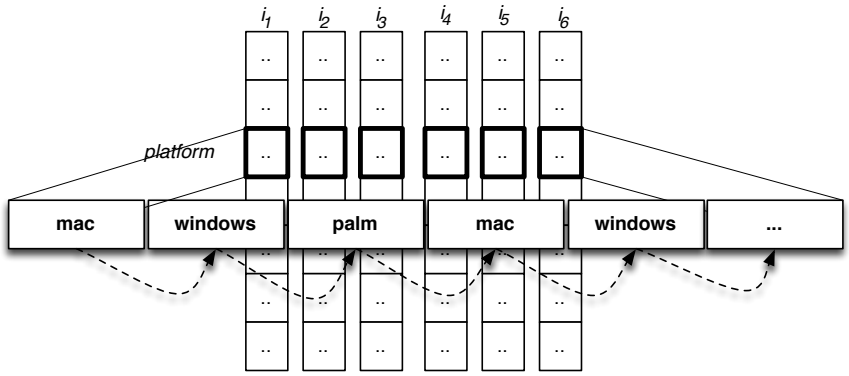

Figure 3.1: Intra-variable interactions for the reasoning variable *platform* (cf. Table 2.1) among six environment instances.

Environment instances and value interactions reflect the environment state at a single moment. Temporality on the other hand, is considered through a sequence of environment instances. The occurrence of the environment instances over time (change from one instance to another) corresponds to a transition. Among transitions, the value of each single reasoning variable fluctuates and produces interactions that I refer to as *intra-variable interactions* because they consider the values of a single reasoning variable. Figure 3.1, illustrates the intra-variable interactions of the reasoning variable *platform* among six instances. In the case of intra-variable interactions, the number of interacting values is infinite. It depends on the number of environment instances considered in these interactions (equivalent to the reasoning variable change rate). For example, if we consider two instances (interaction between two values of the same variable), the reasoning variable *platform* may produce $2^4 = 16$ intra-variable interactions, if we consider three instances, it may produce $3^4 = 81$ interactions and so on.

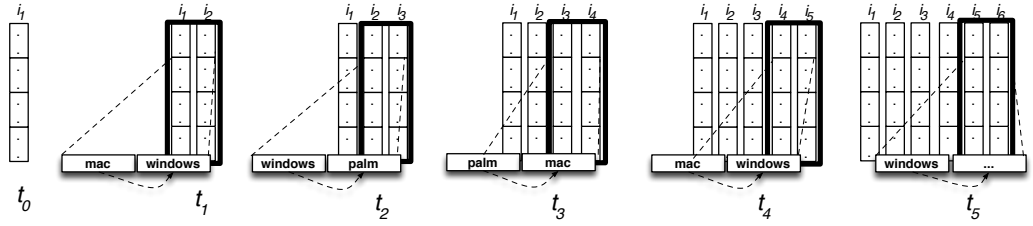

Figure 3.2: Intra-variable interactions considering only two environment instances.

The reason of this variable number of interactions is that, on the one hand, interactions within the same value of a variable (the variable value does not change) are possible and may affect the reasoning. On the other hand, the number of interactions is not fixed because it depends on the time window over which the system reasons. For example, reasoning just about the immediate past implies considering two instances – the current and the immediately past instance. Figure 3.2, illustrates this situation. The time $t_0, t_1, ..., t_5$ indicates the order of occurrence of each instance in the sequence. The reasoning starts considering history after the first instance arrives and continues considering only one instance into the past. The reasoning considers the interactions in Figure 3.1, but only those that happen between the last two values

Interactions may consider the temporality (intra-variable interactions) over several reasoning variables at the same time, they occur between intra-variable interactions in environment transitions. These are inter-variable interactions between intra-variable interactions. For example, Figure 3.3 illustrates the interactions between three intra-variable interactions. Notice that the interaction of these three reasoning variables changing their values in the same period trigger the decision in rule R2 (cf. Listing 2.2).

Self-adaptive systems that make decisions based only on the present environmental condition are fundamentally different from those making decision based o past conditions.

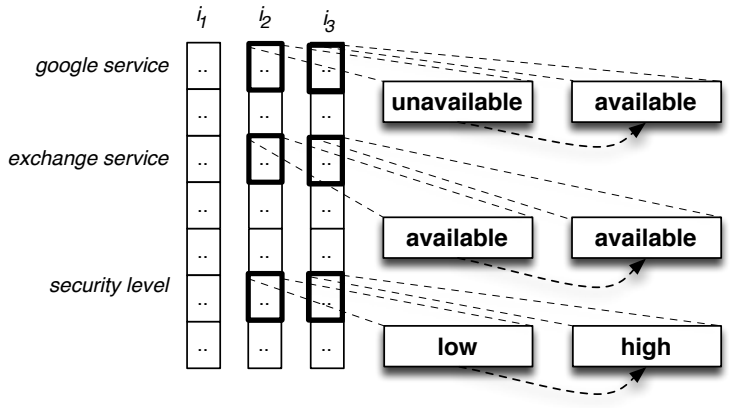

Figure 3.3: Inter-variable interactions between three inter-variable interactions.

The main difference between these two styles of reasoning is the following. On the one hand, when decisions rely only on the present environmental condition, the environmental variations are observed as separate and independent events. Previous environmental conditions do not affect the current decision (because it depends only on the present environmental condition), therefore, the order in which such conditions occur has no importance. Since this reasoning style satisfies the Markov property [78], which states that the next state of a system depends only on the current state, it could be modeled using a Discrete Markov Chain [148] (DMC). The decision making process could be described as a Markov stochastic process, and a DMC could predict the possible (probability distribution) decisions of the system.

On the other hand, when decisions rely on historical and present environmental conditions, the order in which these conditions occur becomes important. The reasoning process considers both inter and intra variable interactions.

These two types of reasoning shape how we model and consider the environment for testing reasoning engines. The first reasoning style requires from testers to consider environmental conditions separately, and to sample only inter-variable interactions. Additionally, since it is possible to model the reasoning process with a DMC, testers have a way to predict the system decisions. The second reasoning style requires from testers to consider the environmental conditions and their occurrence over a time window, and to sample inter and intra variable interactions.

Inter-variable and intra-variable interactions are the core of the reasoning space for reasoning engines that consider history in the decision-making. They are the consequence of the accounting of precise values representing precise environmental states, and streams of values representing change over time. Techniques willing to select portions of the reasoning space must consider these two dimensions of interactions in order to exhibit environment conditions that are likely to trigger all kinds of decisions (right and wrong).

3.2 Mixed Level Covering Arrays

Mixed Level Covering Arrays (MCA) [55] is an extension of (CA) [50], which can handle inter-variable interactions and reduce the size of the reasoning space. The main characteristic of MCA is that it supports the definition of variables with different domain size. CA only supports variables having the same domain size. This puts MCA in good way to sample a large data space such as Cordelia's ACRM system environment. In the following I will introduce MCA, but before introducing it, I present the notion of *combination* upon which MCA is built on.

Definition 4. A k-combination of a finite set S is a subset of k elements of S.

The *k-combinations* specify how to build sets of k elements from an initial set. For example, the *2*-combinations ($k = 2$) or pairs of the reasoning variable *security level* = {*low, medium, high*} (cf. see Table 2.1) are the following:

{ {*low, medium*}, {*low, high*}, {*medium, high*} }

The underlying aim of MCA is constructing an array that contains all the *t*-tuples of inter-variable interactions (interactions between different variables). This will evidence those faults that are produced by interactions between the values of different variables [136]. A MCA is an $N \times k$ array that contains k different variables, where $v_1, v_2, ..., v_k$ are the size of the domain of each variable (number of possible values) and $S_1, S_2, ..., S_k$ are the sets of values of each variable. More precisely, MCA is defined as follows.

Definition 5. *A mixed level* covering array

$$MCA(N; t, k, (v_1, v_2, ..., v_k))$$

is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

1. *Each column i ($1 \leq i \leq k$) contains only elements from a set S_i of size v_i.*
2. *The row of each $N \times t$ sub-array covers all t-tuples of values from the t-combination of columns at least one time.*

The *strength* of a MCA, denoted by t indicates to which extent the covering arrays will cover the inter-variable interactions. A high value of t suggests a better coverage, however, it also implies a larger array. Each column in the MCA represents a particular variable, and each row in a column represents a particular value for that variable. Property (1) ensures that each column contains only values of a particular variable, whereas property (2) ensures the presence of all the t-combinations of inter-variable interactions. Typically, the number of rows N needed to construct a MCA is unknown. Finding the optimal value of N (the smaller number of rows to satisfy the MCA properties) is a challenging problem

Table 3.1: $MCA(18; 2, 11, (2, 2, 2, 2, 2, 2, 2, 2, 4, 3, 2, 2))$.

Exch.	**Goog.**	**Ical**	**Group.**	**Oracle**	**Mem.**	**Use.**	**Plat.**	**Sec.**	**Feed.**	**Conn.**
avail.	*avail.*	*avail.*	*avail.*	*avail.*	*low*	*view*	*win.*	*low*	*like*	*slow*
unav.	*unav.*	*avail.*	*unav.*	*unav.*	*high*	*view*	*mac*	*med*	*dislike*	*slow*
–	–	*avail.*	–	–	–	*view*	*palm*	*high*	–	*slow*
unav.	–	*unav.*	*avail.*	*unav.*	–	*reserve*	*win.*	*med*	–	*slow*
–	*avail.*	*unav.*	*unav.*	–	*low*	*reserve*	*mac*	*high*	*like*	–
avail.	*unav.*	*unav.*	–	*avail.*	*high*	*reserve*	*palm*	*low*	*dislike*	–
–	*unav.*	–	*avail.*	–	*high*	–	*win.*	*high*	*dislike*	*slow*
avail.	–	–	*unav.*	*avail.*	–	–	*mac*	*low*	–	–
unav.	*avail.*	–	–	*unav.*	*low*	–	*palm*	*med*	*like*	–
avail.	*avail.*	*unav.*	*unav.*	*unav.*	*high*	–	*palm*	*high*	–	*fast*
unav.	*unav.*	–	–	–	–	*view*	*win.*	*low*	*like*	*fast*
–	–	*avail.*	*avail.*	*avail.*	*low*	*reserve*	*mac*	*med*	*dislike*	*fast*
avail.	*avail.*	–	–	–	–	*reserve*	*mac*	*med*	*dislike*	–
unav.	*unav.*	*avail.*	*avail.*	*avail.*	*low*	–	*palm*	*high*	–	–
–	–	*unav.*	*unav.*	*unav.*	*high*	*view*	*win.*	*low*	*like*	–
avail.	*avail.*	*avail.*	*avail.*	*avail.*	*low*	*view*	*mwin.*	*low*	*like*	*slow*
unav.	*unav.*	*unav.*	*unav.*	*unav.*	*high*	*reserve*	*mwin.*	*med*	*dislike*	*fast*
–	–	–	–	–	–	–	*mwin.*	*high*	–	–

because it may change according to the disposition of the variable values in the array [50, 34, 104].

We can easily apply CIT techniques to generate testing data for reasoning engines. In this particular case, MCA's columns represent reasoning variables, and rows represent environment instances. Since the covering array is constructed without specifying a particular order, variable transitions are ignored. For example, Table 3.1 presents the MCA for Cordelia's ACRM system environment (cf. Table 2.1) MCA (18;2,11,(2, 2, 2, 2, 2, 2, 2, 4, 3, 2, 2)). In this case, the covering array seeks to optimize the interactions between pairs of reasoning variables ($t = 2$). It contains 18 environment instances that arrange all the inter-variable interactions of pairs of variables. If we compare this with the 6.144 instances needed to exhaustively test the reasoning engine, MCA represents an extensive reduction in the amount of data needed to test Cordelia's ACRM reasoning engine.

3.3 Limits of MCA for sampling the reasoning space

Mixed level covering array (MCA) is effective at reducing the number of inter-variable interactions to test and at finding faults due to these interactions [136]. In the previous section I showed that inter-variable interactions are important in the reasoning process since they trigger decisions over particular values for reasoning variables. Yet, MCA does not address all the dimensions of sampling the reasoning space. MCA fails to handle the tem-

poral dimension of reasoning – intra-variable interactions. Since it produces environment instances without specifying a particular order, it disregards the importance of environment transitions.

Although MCA does not ensure an admissible sampling of the reasoning space, its underlying theory merits to be extended. Such extension must consider the interactions that come with temporality – intra-variable interactions. In the next section I present an extension to MCA, which consider inter-variable and intra-variable interactions to generate covering arrays.

3.4 Multi-Dimensional Covering Arrays

I introduce *multi-dimensional covering arrays* (MDCA) as an extension to MCA meant to handle inter-variable and intra-variable interactions at the same time. This technique is *multi-dimensional* because it is capable of handling several dimensions of interactions and integrates them into a single data selection process. Before introducing MDCA, I introduce the concept of *u*-transition upon which I build MDCA. The *u*-transitions represent the different ways that intra-variable interactions can occur for a given reasoning variable.

Definition 6. A u-transition of a finite set S is the set of all ordered u-tuples that can be generated with u elements from S.

Notice that the u-transition is not a permutation of a set in the usual sense of the term. It specifies how to build sets of u elements from an initial set, and not the ways to swap u elements in a set. For example, the 2-transitions ($u = 2$) of the reasoning variable *security level* consists of the following elements:

{ <*low, medium*>, <*low, high*>, <*medium, high*>, <*high, medium*>, <*high, low*>, <*medium, low*>, <*low, low*>, <*medium, medium*>, <*high, high*> }

The u-transitions contain all the possible intra-variable interactions for a particular reasoning variable. Contrarily to the u-combinations (cf. Definition 4 in Section 3.2), the u-transitions consider the order in which elements are selected and can contain repeated elements. For example, the pair <*low, medium*> is different from the pair <*medium, low*> and the pair <*medium, medium*> is a valid element of the *2*-transition.

The aim of MDCA is covering all the inter-variable and intra-variable interactions. This should reveal those faults in the reasoning that are caused not only by interactions between different variable values, but also by the different transitions of each variable.

Similarly to a MCA, a MDCA is a $k \times N$ array that contains k different variables, where $v_1, v_2, ..., v_k$ are the cardinalities (number of possible values) and $S_1, S_2, ..., S_k$ are the values of each variable. The *strength* of a MDCA, denoted by t, indicates the coverage of transition interactions (notice that the coverage of transition interactions implies the coverage of inter-variable interactions), whereas the *chaining strength*, denoted by u indicates

the coverage of the intra-variable interactions. Each row in the MDCA represents a particular variable, and each column represent a particular value for that variable.

Definition 7. *A multi-dimensional* covering array

$$MDCA(N; u, t, k, (v_1, v_2, ..., v_k))$$

is an $k \times N$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

1. *Each row i ($1 \leq i \leq k$) contains only elements from a set S_i of size v_i.*
2. *Each u-tuple of u-consecutive columns in a row i ($1 \leq i \leq k$) contains only elements from the u-transitions of S_i.*
3. *The u-consecutive columns of each $t \times N$ sub-array cover all t-tuples of values from t-combinations of P_i sets (i are the indexes of the selected t-rows) at least once, where P_i are the u-transitions of S_i.*

Property (1), ensures that each row contains elements of the variable it represents. Property (2), ensures that the u-consecutive columns contain the intra-variable transitions of that variable. Property (3), ensures that all the array covers all the *t*-tuples of variable transitions. Notice that MDCA constructs u-tuples of values, which represent the intra-variable interactions out of consecutive columns. This is because MDCA specifies an order of appearance for columns.

	1	2	3	4	5	6	7	8	9	10	11	12	13
Conn. Speed	*slow*	*fast*	*slow*	*slow*	*fast*	*slow*	*slow*	*slow*	*slow*	*slow*	*fast*	*fast*	*fast*
Sec. level	*high*	*low*	*high*	*high*	*med*	*high*	*med*	*high*	*low*	*high*	*high*	*low*	*med*
Use type	*view*	*resv*	*view*	*view*	*view*	*view*	*resv*	*view*	*view*	*view*	*resv*	*resv*	*view*

	14	15	16	17	18	19	20	21	22	23	24	25	26
Conn. Speed	*fast*	*fast*	*slow*	*fast*	*slow*	*slow*	*fast*	*slow*	*slow*	*fast*	*slow*	*slow*	*fast*
Sec. level	*low*	*high*	*med*	*med*	*med*	*med*	*low*	*med*	*low*	*low*	*low*	*low*	*med*
Use type	*view*	*resv*	*resv*	*view*	*resv*	*resv*	*resv*	*resv*	*view*	*resv*	*resv*	*view*	*view*

	27	28	29	30	31	32	33	34	35	36	37	38	39
Conn. Speed	*fast*	*fast*	*slow*	*fast*	*fast*	*slow*	*slow*	*fast*	*slow*	*fast*	*fast*	*fast*	*fast*
Sec. level	*high*	*med*	*low*	*high*	*high*	*low*	*med*	*high*	*high*	*low*	*low*	*med*	*med*
Use type	*resv*	*view*	*resv*	*resv*	*resv*	*view*	*resv*	*resv*	*view*	*view*	*view*	*view*	*view*

Figure 3.4: $MDCA(39; 2, 2, 3, (2, 3, 2))$

Notice that each row in the MDCA represents a flow of values for a particular reasoning variable, and each column represents an environment instance. The order of appearance of each instance is important, since this order represents the different variable transitions.

For example, Figure 3.4 presents the MDCA for three reasoning variables ($k = 3$): *use type* ($v_1 = 2$), *security level* ($v_2 = 3$), and *connection speed* ($v_3 = 2$). It contains 39 environment instances, each pair of instances ($u = 2$) contains a variable transition, and each pair of rows ($t = 2$) contains all the pairs of reasoning variables with change rate 2. Notice that I present this example over 3 reasoning variables because the MDCA for all the 11 reasoning variables of Table 2.1 would not fit into the page.

Comparing the size of MCA and MDCA sets, MCA produces 18 environment instances that cover all the inter-variable interactions. On the other hand, MDCA produces 141 environment instances that cover all the inter-variable and intra-variable interactions. Even if we consider that MDCA produces sets 8 times larger than MCA (in this particular case), the set size is well rewarded by the degree of coverage it provides. Besides, it is a significant reduction compared to the total number of instances needed to exhaustively exploring the whole reasoning space of reasoning variables with change rate 2. That is, 142 against about 37×10^6 instances.

MDCA is good at exploring large reasoning spaces because: (1) **It proposes at least the same coverage of inter-variable interactions as MCA**; and, (2) **it answers the requirements of the reasoning space by covering the intra-variable interactions**. MDCA covers the different transitions of each reasoning variable and the different interactions between these transitions.

Covering transitions and their combinations increases the probability of triggering decision-making that relies on transitions and combinations of transitions. Notice that, the covering array's *strength* describes the level of coverage of transition interactions (which supersedes the inter-variable interactions), whereas the *chaining strength* describes the level of coverage of intra-variable interactions. In section 3.6, I demonstrate through a series of experiments the validity of this claim.

3.5 Constructing MDCAs

The problem of constructing MDCAs consists in ordering the array elements in the most convenient way. That is, arranging the elements inside an array in such a way that it satisfies all the properties in Definition 7. This problem is similar to constructing covering arrays, which is a NP-complete problem [226, 204]. Basically, it consists in finding the smallest array that satisfies all the MDCA's properties. In other words, given a $k \times n$ array, (1) arrange its elements in such a way that the array contains all the t-tuples of transitions with change rate u (Property 3 of Definition 7); and (2) find n close to N, the minimum number of columns. Constructing MDCAs is complex because the arrangement of a particular cell (element) in the array may affect the disposition of other cells in the whole array; therefore, the number of combinations for the cells disposition in the array is huge.

This problem can be solved using different techniques such as *backtracking*, *meta-heuristics*, etc. In this particular case I use functional optimization and two meta-heuristic

techniques to address the problem: *genetic algorithm* [96] and *bacteriologic algorithm* [12]. Additionally, I introduce a third meta-heuristic that combines elements from genetic and bacteriologic algorithms.

Functional optimization [232] uses a mathematical function that quantifies how good or bad a solution is (or in this case an array). In meta-heuristic search, such function has the underlying goal of guiding the search for the best possible solution to a problem. It this case, the optimization function indicates how far is the arrangement of elements in the current solution with respect to the expected optimal solution. This will guide the search to solutions whose arrangement better satisfies the MDCA's properties. Notice that I use the optimization function to guide the search for the arrangement of elements that contains the greater number of tuples, whereas I let the meta-heuristics to search for the optimal length. I define the optimization function for constructing MDCAs as follows.

Definition 8. *The maximization function for X, a $k \times n$, u the* chaining strength*, and t the* strength *is:*

$$f(X, u, t) = \text{\# of different } t\text{-tuples of transitions with change rate } u \text{ in } X.$$

The maximization function $f(X, u, t)$ will guide the meta-heuristic search towards solutions that contain a large number of tuple. The greater the number of tuples, the closer the solution is to satisfy the properties in Definition 7.

Notice that in the following, I use $f(X)$ and $f(X, u, t)$ indistinctly. It is possible to calculate $f(X, u, t)$ in several ways, I calculate it by counting for each $k \times u$ sub-array (formed by u-consecutive columns) the number of different t-tuples in the array. In Annex A, I provide the pseudo code for counting the number of tuple in a $k \times n$ array X.

3.5.1 Optimal value

The objective of the meta-heuristic search is to increase $f(X, u, t)$'s value. Since the covering arrays must hold a fixed number of t-tuples, it is possible to calculate the expected value e for $f(X, u, t)$. I define the expected value e as follows.

Definition 9. *The expected number e of t-tuples of u-transitions for an MDCA over k variables, with $v_0, v_1, v_2, ..., v_k$ elements each, a* strength *t, and* chaining strength *u:*

$$e = \sum_{i=1}^{m} \prod_{j=1}^{t} p_{c_{(i,j)}}, \text{ where}$$

$$\begin{array}{lcl} p_i & = & v_i^u \\ m & = & \frac{k!}{t!(k-t)!} \\ c_{m \times k} & = & u\text{-combinations of } \{1, 2, ..., k\} \end{array}$$

Notice that the expected value e only indicates the number of t-tuples and not the size of the array. For example, the value of e for the MDCA in figure 3.4 is 88, whereas the size of the array N is 39.

Since we know e, when $e - f(X, u, t) = 0$, X is an MDCA with *strength* t and *chaining strength* u. Therefore, this is the stopping criterion that will rule the meta-heuristic search I propose to construct MDCAs.

3.5.2 Genetic Algorithm

A genetic algorithm [96] (GA) is a search technique to find an exact or approximate solution to optimization and search problems. It is inspired by the evolution of the species and the survival of the fittest individual.

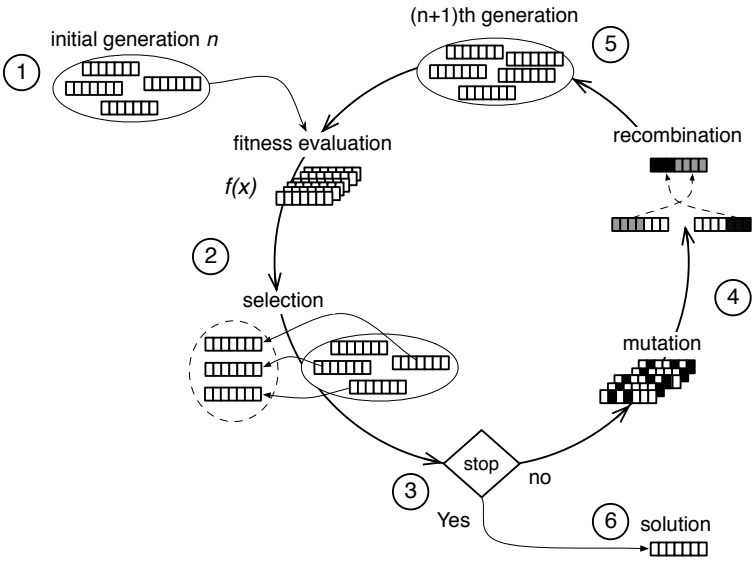

Figure 3.5: Flowchart of a genetic algorithm.

GAs are implemented in a computer simulation in which a population of abstract representations (called chromosomes or the genotype of the genome) of candidate solutions (individuals) to an optimization problem evolves toward better solutions. Figure 3.5, presents the flow of a genetic algorithm. Initially, the evolution usually starts from an initial population (1) of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated (2) and multiple individuals are stochastically selected from the current population (based on their fitness), and modified by recombination and mutation (4) to form a new population. The new population (5) is then used in the next iteration of the algorithm. Commonly, the algorithm

terminates (3) when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. The result of the evolution corresponds to the best solution of the problem that the GA could find (6).

Adaptation to MDCA

In order to use a GA to construct MDCAs, I have encoded each individual as a $k \times n$ array on $V = \sum_{i=1}^{k} v_i$ symbols. Therefore, a population is a collection of $k \times N$ arrays. The fitness of each individual is determined using the optimization function $f(X, u, t)$, and the stopping criterion $e - f(X, u, t) = 0$ is applied in addition to a maximum number of generation. I define the operations that are applied in each generation as follows.

Definition 10. *The mutation and recombination of a $k \times N$ array on $V = \sum_{i=1}^{k} v_i$ symbols are:*

1. Mutation *of an individual X, consists in changing the particular value of a randomly selected row i and column j by a random value v ($1 \leq v \leq v_i$), $X_{(i,j)} = v$.*
2. Recombination *of two individuals X', X'' consists in selecting a random column j and replace in X' all the columns $j + 1, ..., n$ by the same columns of X''.*

Notice that mutation and recombination are applied at a predefined rate. These, and several other parameters such as the population size, initial population size, and maximum number of generations can affect the performance of the GA. For example, a mutation rate of 0.5 means that out of 10 individuals, 5 stochastically selected are going to be mutated. The population size indicates how many individual are going to be selected out of the total population. The initial population size indicates how many individuals may conform the population initially fed into the algorithm. Finally, the maximum number of generations indicates how many times the population can evolve.

The benefit proposed by GA to the construction of MDCAs is the fast convergence towards an optimal solution when the number of columns (size of n) is close to the optimal. Yet, the major drawback comes from the fact the n is fixed. When the value of n is under the minimum, the solution fitness $f(X)$ will never reach its optimal value e. When the value of n is over the minimum, $f(X)$ will reach e faster, but the solution will be of lower quality.

3.5.3 Bacteriologic Algorithm

A bacteriologic algorithm [12] (BA) is an original adaptation of GA as described in [96]. The general idea is that a population of bacteria is able to adapt itself to a given environment. If they spread in a new stable environment they will reproduce themselves so that they fit better and better to the environment. At each generation, the bacteria are slightly altered and, when a new bacterium fits well a particular part of the environment

it is memorized. The process ends when the set of bacteria has completely colonized the environment.

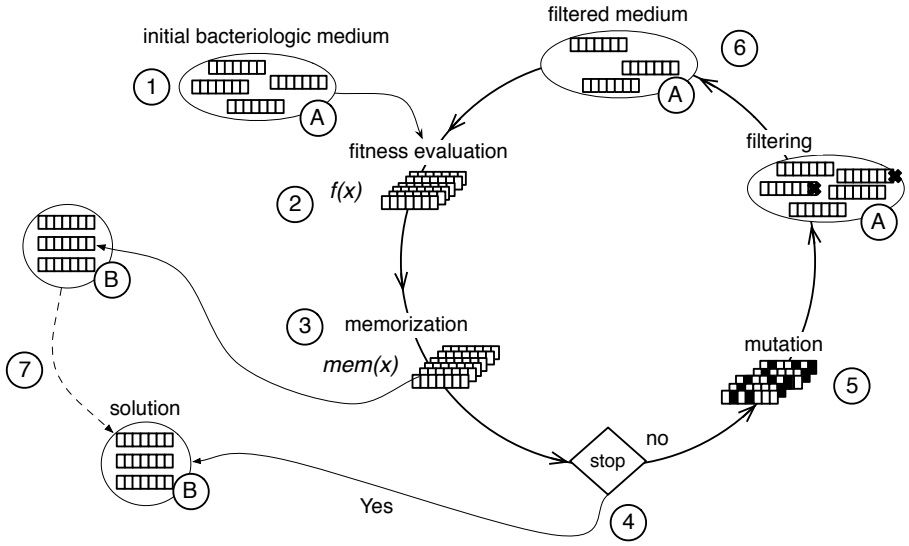

Figure 3.6: Flowchart of a bacteriologic algorithm.

Figure 3.6, presents the flowchart of a BA. Along the execution there are two sets, the solution set (B) that is being built, and the bacteriologic medium (A) or a set of potential bacteria. It starts with an initial set of bacteria (1), and its evolution consists in series of mutations on bacteria to explore the whole scope of solutions. The final set is built incrementally by adding bacteria that can improve the quality of the set. First, the fitness of each bacterium in the bacteriologic medium is computed (2). Second, the bacteria capable of improving the solution set are then memorized into the solution set (3). Third, in order to explore new solution elements, each bacterium in the bacteriologic medium is modified through mutation (5). Fourth, the bacteriologic medium is filtered out of bad and useless bacteria (6), this keeps the medium size under control. Finally, after each memorization the solution set is evaluated (4) to check whether the algorithm should stop. Commonly, the algorithm terminates after a number of generations, when a minimum fitness value is reached by the solution set, or the fitness has not changed for a number of generations. The resulting solution corresponds to the best solution the BA could find (7).

Adaptation to MDCA

The representation of the problem in this case is slightly different from the one used by the GA. A Bacterium is a vector of k elements, each one containing a variable value. The

final solution set is a $k \times n$ array, composed of n vectors of k elements. The fitness function of the solution set is determined by the optimization function $f(X, u, t)$. Furthermore, BA uses two fitness functions. One, $f(X, u, t)$ is used to verify whether the solution has reached the optimal. The other, $f_r(X, u, t)$ is used to evaluate the contribution of each particular bacterium and determine whether it should be memorized. In this case, the relative fitness corresponds to the contribution of each bacterium to the solution set. We define the relative fitness function as follows.

Definition 11. *The relative fitness function f_r of a bacterium y, and a solution set $X_{k\times n}$ is:*

$$f_r(X, y, u, t) = f(X \cup y, u, t) - f(X, u, t)$$

Each bacterium that makes a contribution to the solution set is memorized, and each bacterium that does not, is removed from the medium. We define the mutation of a bacterium as follows.

Definition 12. *The* Mutation *of a vector of length k on $V = \sum_{i=1}^{k} v_i$ symbols (bacterium y) consists in selecting at random a value i from k, and changing it by a random value v ($1 \leq v \leq v_i$).*

Notice that several parameters can influence the performance of BA. Such parameters are the mutation rate, the memorization threshold, the maximum number of generations, the bacterium' maximum age (number of turns in the medium), and the medium size.

The main benefit of BA is that the number of columns (size of n) can vary. Since the final solution set is constructed incrementally, the number of bacteria (columns) in the solution may vary according to their fitness contribution. Yet, this is also a drawback. Depending on the initial set of bacteria, it is possible that the final solution set actually reaches e with too many bacteria (many more than the minimal). Furthermore, it can get stuck in a local solution if no bacteria can increase the overall value of $f(X)$, or if more than a single bacterium is need to increase the value of $f(X)$.

3.5.4 Hybrid Algorithm

The Hybrid Algorithm (HA), is a modification of the BA to include a local optimization through a GA. Such optimization is performed after adding new bacteria to the solution set. This algorithm also performs a solution *fixing* when a solution set gets stuck into local optima. This fixing consists in adding a new t-tuple of bacteria to the solution set, which contains one or more t-tuple of transitions missing in the solution set. These t-tuples are generated by enumerating all the t-tuples and picking one or more of them that are not in the solution set. This automatically increases the value of $f(X)$ and allows the algorithm to explore other solutions.

Figure 3.7, presents the flowchart of an HA. It is essentially a BA, however, just after memorization a GA *optimizes* the solution set. The solution set is used to generate the

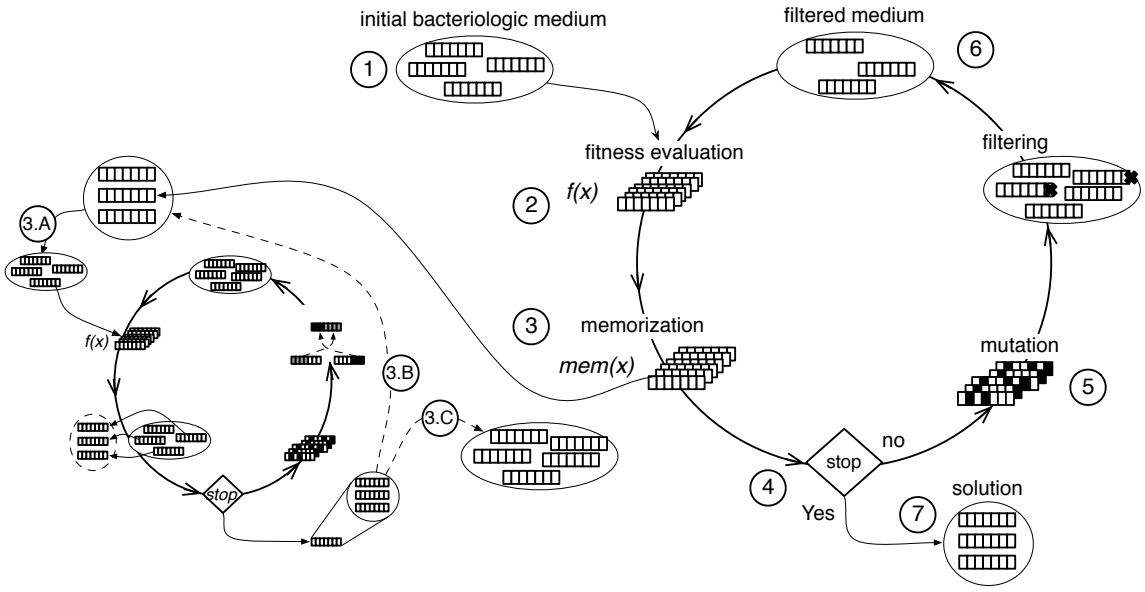

Figure 3.7: Flowchart of an hybrid algorithm.

GA's initial population (3.A), and the evolution takes place. If a better solution is found, it replaces the old solution set (3.B). Additionally, new bacteria are inserted into the bacteriologic medium: those that optimize the solution set and were not present in the medium (3.C). Everything then continues analogously to a BA.

HA possesses the benefits of GA and BA. It is capable of incrementally constructing a solution and exploring local solutions at the same time. This ensures that the solution will never be suboptimal and reduces the chances of having a solution set whose length is larger that the optimal. The main drawback of HA is the large number of iterations it may use to find a solution, though it may find a solution with smaller value of n.

3.6 Experimental Evaluation

In this section I present the empirical evaluation of the feasibility and efficiency of MDCA. First, I evaluate the feasibility of MDCA by comparing the three algorithms I previously presented GA, BA, and HA. My findings show that HA is the best-suited technique to construct MDCAs with minimal length and satisfying the MDCA properties. Second, I evaluate the efficiency of MDCA by studying the coverage of reasoning conditions it provides over three experimental subjects. I compare the MDCA performance against two well-known sampling techniques: MCA with *strength* 2 (known as pairwise), and random sampling of environment instances.

3.6.1 Experimental subjects

I evaluate the effectiveness of MDCA over three experimental subjects. These range in size from large to small, and implementation from Drools to Java. Notice that each of these subjects uses rule based reasoning as basis for reasoning, however, MDCA and the other data generation techniques are applicable to any reasoning technology.

The first (I) experimental subject is the Drools implementation of Cordelia's ACRM system. This implementation contains 40 reasoning rules, out of which 21 have conditions over transitions and tuples of transitions. Notice that some conditions aggregate several variable values to make decision. The second (II) experimental subject is the Java implementation of Cordelia's ACRM system. This implementation uses a custom reasoning engine and contains 96 reasoning rules (expressed as *if-then-else* Java statements equivalent to about 860 LOC[†]), out of which 22 have conditions over transitions and tuples of transitions. Notice that the first and the second subject do not have equivalent reasoning in every case. Technical limitations led the developers of these systems to make different decision regarding the reasoning process. The complex event-processing feature of Drools are not available in Java, therefore developers implemented custom solutions. For instance, for the first subject, developers use the Drools capabilities to state temporal rules, whereas for the second subject, temporality is represented by a data structure (array) and temporal rules refer directly to that data structure. Another difference is that in the first subject, the Drools engine uses a Java base backend, but defines its own rule language, whereas the second subject is implemented in Java only. The consequence of these differences is that given a sequence of environmental conditions, the first and second subjects will make different decisions. Nonetheless, despite that these two implementations have different behavior, they share the same representation of the environment.

The third (III) experimental subject is an *adaptive web server* [41], which changes its internal properties such as cache size according to its load. Its environment is composed of three variables *request density* (number of requests), *request dispersion* (distribution of request over time), and *file diversity* (number of different files). Each of these variables can have three values *low*, *medium*, and *high*. Notice that the server's reasoning engine uses rules that quantify the variable using fuzzy logic. It comprises 17 reasoning rules, none of these with conditions over transitions.

3.6.2 Research questions

In this experimental evaluation, I seek to answer two research questions. These questions and their answers are intended to provide helpful insight about the construction of MDCA and their efficiency to cover the different reasoning possibilities.

Q1 *Is it possible to construct MDCAs? Which is the best way to construct MDCAs?* The answer to this question will reveal whether it is possible or not to construct an $N \times k$

array that satisfies the conditions in definition 7. Furthermore, the best procedure to construct MDCA will generate arrays as small as possible.

Q2 *Are MDCAs efficient at covering reasoning conditions? Are they capable of providing better coverage compared to other techniques?* The answer to this question is essential to ratify the contribution of *MDCA*. If MDCA is capable of covering a major portion of the conditions used by reasoning engines to make decisions, then it will prove being suitable for assessing the correctness of reasoning engines' decisions. Furthermore, if it performs better than other techniques at covering reasoning conditions, then it will prove its worthiness with respect to those techniques.

3.6.3 Experimental SetUp

The experimental setup corresponds to the data, configuration, and parameters I use to perform the experiments. This information defines the context upon which the experimental results are valid, and conclusions can be drawn. In the following I introduce the experimental setup.

Experimental data

Regarding the construction of MDCAs, with each algorithm (GA, BA, and HA) I constructed 2 sets of 50 MDCAs (sets of instances). Notice that, there are 2 sets instead of 3 (3 subjects). The rationale for this is that the experimental subjects I and II use the environment representation.

Regarding the construction of pairwise and random sets of instances, I constructed 2 groups of 50 sets of instances satisfying the MCA properties (cf. Section 3.2). To generate such data I use a third party tool [1]. In order to perform a *fair* comparison, I randomly mixed the instances produced by the pairwise to equate the size and variance of the best MDCA construction algorithm. Furthermore, I also randomly generate 2 groups of 50 sets of instances (each variable value is selected randomly) that equate in size and variance the best MDCA construction algorithm.

Experimental parameters

Since most reasoning rules in the case studies reason over at most reasoning variables with change rate 2 (a minor number have change rate 3), I deliberately fixed the MDCA's *chaining strength* to 2. Furthermore, conditions are declared on at most three different variables, being more than 80% of the conditions declared over two or less variables. Thus, again, I deliberately fixed the MDCA and MCA *strength* to 2. Regarding the meta-heuristic

1. http://www.mcdowella.demon.co.uk/allPairs.html

construction of MDCA, I parameterized each construction algorithm as follows:

GA I set the mutation and combination rate to 0.5, the number of maximum generations to 500, the population to 100 individuals, and the initial population to 30 individuals (randomly generated).

BA I set the mutation rate to 0.5, the number of maximum generations to 500, the initial bacteria population to 10 (randomly generated), the medium size to 100 bacteria, and the bacterium maximum age to 10.

HA I use the same parameterizations described for BA and GA.

Evaluation criteria

To answer the first question, I compare the MDCAs constructed by each algorithm BA, GA, and HA. This comparison is based on three criteria: (1) number of *generations* that the algorithms take to reach the expected fitness value; and (2), the average size and dispersion of the constructed MDCA. To answer the second question, I compare the coverage of MDCAs, MCA, and RAND over the three experimental subjects. For the first and third experimental subject, I use the coverage of rule's conditions as a metric to evaluate the effectiveness of the data to cover the different reasoning paths. For the second experimental subject, I use the coverage of executable LOC as a metric to evaluate the effectiveness of the data to execute the different parts of the program (reasoning paths).

It is important to notice that the use of rule conditions coverage is a metric applicable only to rule based reasoning. With this metric I aim at comparing the likeliness of each criteria to ultimately find faults. On this purpose, I make the hypothesis that the more rules decisions can be covered, the better are the chances to find faults, especially those due to the interactions described in Section 3.1. Additionally, I make the hypothesis that since MDCA is an exploration technique, which is independent from the reasoning technique, it may provide similar results with other reasoning techniques.

3.6.4 GA v/s BA v/s Hybrid

Figure 3.8 shows 2 different plots comparing the performance of GA (dashed black line), BA (dotted red line), and HA (solid blue line). The curves were constructed using the average value of 50 algorithm executions in each case. Table 3.2, summarizes statistically relevant data for these results.

The first plot (a), shows a comparison of performances in terms of number of generations versus fitness. From the plot we observe that the three algorithms are capable of constructing MDCAs, however, we need to consider that since GA cannot compute the solution length dynamically, we need to fix it manually. This greatly affects the algorithm's performance because the more elements an individual contains (solution length), the faster

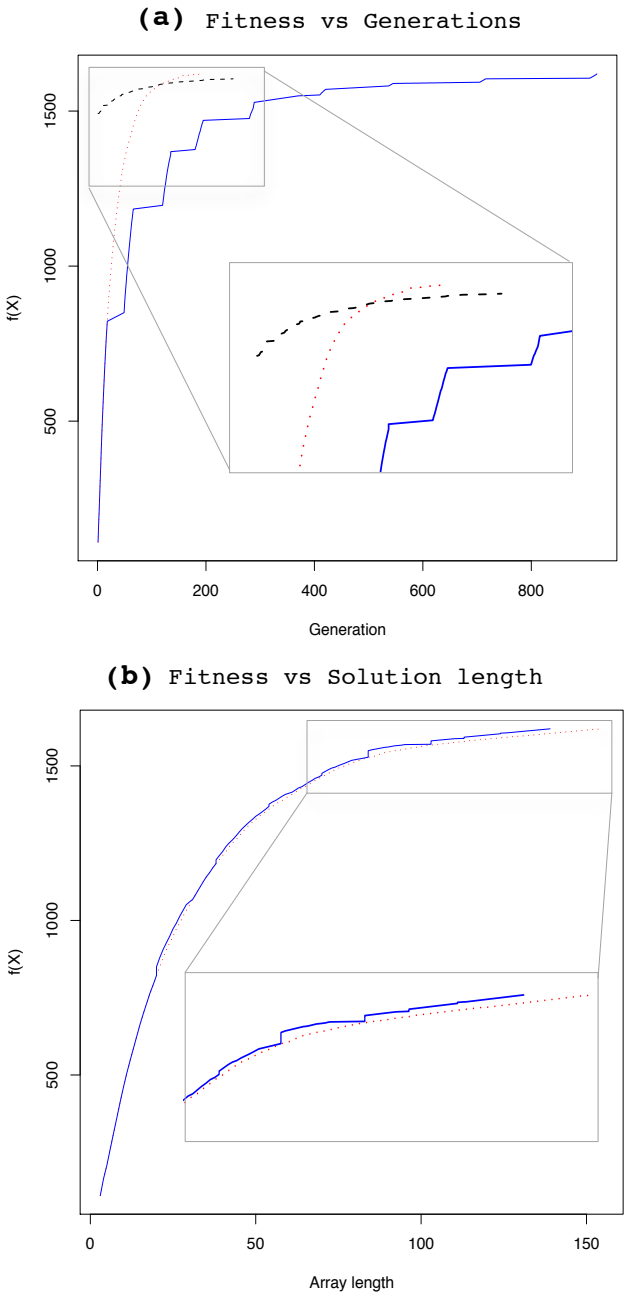

Figure 3.8: Two plots comparing the performance of GA, BA, and HA. The length of GA, BA, and HA are represented respectively by the dashed black line, the dotted red line, the solid blue line.

		Min	Q1	Median	Mean	Q3	Max
GA	Iterations	250	252	258	260	270	300
	Length	145	145	145	145	145	145
BA	Iterations	170	175	187	189	230	260
	Length	142	151	153	152	157	160
HA	Iterations	900	910	920	921	930	950
	Length	139	140	141	141	143	145

Table 3.2: Statistical data from iterations and length of GA, BA, and HA.

the algorithm converges to a solution. Nevertheless, when the solution length is fixed to the minimal, the algorithm takes more generations to converge. From this observation, we deduce that GA is not the best-fitted algorithm to solve this problem. Also, We notice that HA takes considerably more generations than BA to converge (HA: 921 versus BA: 189 cf. Table 3.2). This is because BA only adds elements to the solution set, whereas HA optimizes the solution set each time (trying to get the most of if). Since the generations of HA consider the generations of the underlying BA and the underlying GA, its overall generation number is larger. This represents an advantage point for BA.

Since BA and HA can handle variable solution length, this is the next comparison point. The second plot (b), shows a comparison based on the solution length versus fitness. From the plot, we observe that HA has small punctual improvements in particular points (solution lengths), that make the overall HA's solution length shorter than BA's solution length (HA: 141 versus BA: 152 cf. Table 3.2)). This is because of the HA local search (through a GA, cf. see Section 3.5.4), which allows it to converge towards solutions with shorter solution length. Since shorter solution sets may imply fewer scenarios to play, we prefer them. This represents an advantage point for HA.

The decisive point to judge which algorithm is better, is comparing each algorithm' solution length variation. The box-plot in Figure 3.9 graphically illustrates the variation of the solution length. A box-plot [218] is a type of graph used to display patterns of quantitative data. It is composed of a box, which goes from the first quartile (Q1) to the third quartile (Q3). Within the box, the horizontal line at the center corresponds to the median of the data set. Two vertical lines, called whiskers, extend from the front and back of the box. The front whisker goes from Q1 to the smallest (min) non-outlier in the data set, and the back whisker goes from Q3 to the largest (max) non-outlier.

From the plot, we observe that the solutions produced by HA tend not only to be shorter, but also to be more stable. BA produces longer solutions with much variation. This is because the initial solution set and the evolution of bacteria strongly affected the BA's final solution set. Contrarily, the initial solution has less influence in the HA's final solution set. Since HA locally optimizes the solution set, it is capable of escaping local optima, and avoid adding more element to the solution set.

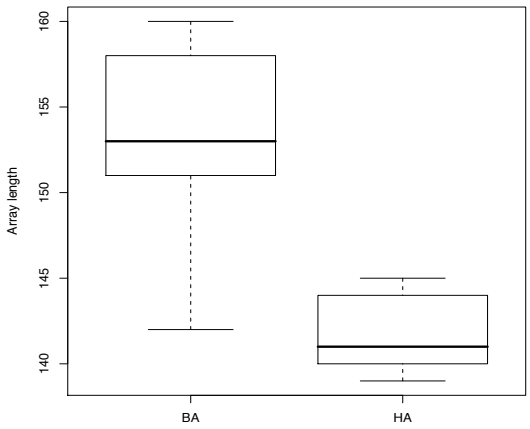

Figure 3.9: Two plots comparing the performance of BA and HA.

Given the empirical evidence, the **best MCDA construction algorithm is HA**.

3.6.5 Comparing generation sets

We have measured the coverage of each of the three data sets (50 sequences of instances) MDCA, MCA, and RAND over our three experimental subjects. Notice that Subjects I and II are the realizations of the same system but with different formalisms, which implies a different number of rules.

Figure 3.10 shows three box-plots graphically displaying the coverage results for each experimental subject. The coverage results serve to compare the performance of the different data sets. From the box-plots, we can observe:

(1) MDCA provides a better coverage than MCA and RAND in subjects I and II (average coverage in subject I: 33 rules, and in subject II: 751 LOC). We explain this by the fact that these subjects define a major part of their reasoning conditions over reasoning variables with change rate 2. Since MDCA ensures the coverage of all transitions with change rate 2 (and their pairs), the only reasoning variables left to cover are those with change rate 3 or more. The reasoning conditions over reasoning variables with change rate 3 or more are few, 9 in subject I, and 12 in subject II (equivalent to about 109 LOC). The MDCA covers some of these transitions, but in general they prevent MDCA to cover all the reasoning conditions. A MDCA set with higher strength and chaining strength may cover all the reasoning conditions.

(2) MDCA not only provides better coverage, but also provides more stable coverage. Since MDCA ensures the coverage of all the reasoning variables with change rate 2 (and their pairs), the only variation occurs in the coverage of reasoning conditions

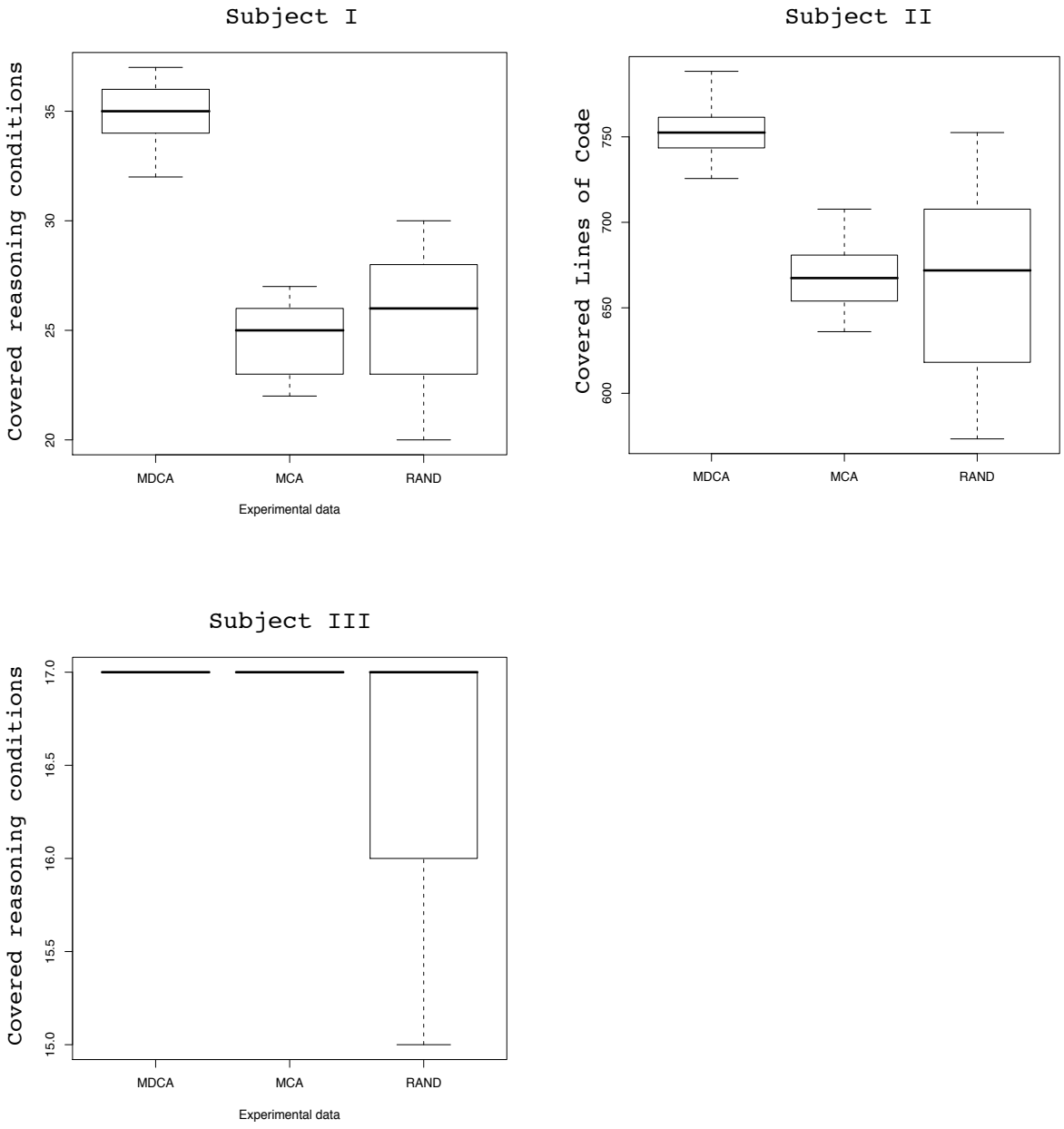

Figure 3.10: Three plots comparing the coverage of reasoning conditions by MDCA, MCA, and RAND over the three experimental subjects

over reasoning variables with change rate 3 or more. The coverage of these conditions is erratic because MDCA cannot always ensure their coverage.

(3) The MCA provides a poor but more stable coverage than RAND. The reason of this is that MCA ensures the coverage of inter-variable interactions (it does not specify any order, and do not care about transitions). Thus, this ensures the coverage of all the reasoning conditions declared over single variable values (and their pairs). The variation in the coverage comes from reasoning conditions over reasoning variables with change rate 2 or more. Ergo, MCA provides a poor coverage in subjects I and II.

(4) RAND provides an overall better coverage than MCA, but with more variation.

(5) The coverage of MDCA and MCA in subject III is 100%. The underlying reason for this is that the experimental subject contains very few variables, and the reasoning conditions are declared only over particular variable values. These values are easily covered by both MCA and MDCA. Regarding RAND, in some cases (out of 50) a variable value is missing, causing the reasoning conditions coverage to vary.

These 5 observations provide enough evidence to answer question 2. MDCA provides at least a coverage of 82% of the rules in subject I, of 98% of the LOC in subject II, and 100% of the rules in subject III. This suggests that MDCA ensures a coverage level of reasoning conditions. We expect that with higher *strength* and *chaining strength*, the coverage level will increase. For example, with a *chaining strength* 3, MDCA should ensure the coverage of 100% of the reasoning conditions in subject I. The evidence also suggests that MDCA also provides better coverage than MCA and RAND. Nevertheless, the evidence provided by subject III suggest that MCA and MDCA have comparable performance if the reasoning does not consider history. This seems reasonable, since MDCA is an extension of MCA to handle the combinatorial explosion produced by history. Furthermore, MCA would cover the same number of conditions that MDCA with less data.

3.6.6 Threats to validity

There exists no perfect data, or perfectly trustable analysis results, and this study is not an exception. For this reason we identify the construction, internal and external threats to validity for this study.

Internal threats lie on the source of the empirical data. I have constructed MDCAs with the tools we think are the best fitted. Nevertheless, if the MDCA sets are not optimal, they may contain more data that will increase the coverage rate in the case studies without really reflecting the MDCA's aim. I also selected our experimental subject considering that the rule they contain cover as best as possible the reasoning spectrum. We cannot ensure that the subjects' conditions are diverse enough to represent the entire reasoning spectrum. If the reasoning rules and variables of each experimental subject are more likely to be covered by MDCA and no other technique, then the good result we obtained will not represent the

reality.

Construction threats lie in the way we define our metrics and their measurement. One threat comes from the generation of MCAs. Since we use a third party algorithm, we cannot ensure that the solution it produces is always optimal. Non-optimal MCA can affect negatively the reasoning conditions coverage. Furthermore, the shuffle of MCA elements can bias the results increasing the coverage it provides. I have compared MDCA construction using several criteria, including the different empirical subjects. I think that the algorithm is the best for constructing MDCAs, however, I cannot ensure that it will not perform poorly in presence of a very larger number of environmental variables.

External threats lie on the statistical significance of our study. I evaluated the coverage of MDCA over three experimental subjects that I considered proper given the source and nature of the system. They come from the industrial partner of a European research project [187]. Nevertheless, it is possible that they only reflect the trends on a very particular application field. I cannot ensure that these subjects reflect the trends of all application fields.

3.7 Discussion

Ensuring that reasoning engines make the correct decision is challenging because it involves selecting complex testing data over a very large data space. Mixed level covering array, a state of the art black-box testing technique helps engineers to deal with the large size of the reasoning space. It consists in sampling a limited number of interactions between the variables that constitute the reasoning space. This ensures that at least, the interactions between combinations of variables are covered, and that if there exist faults related to these interactions they will be found. Nonetheless, MCA does not cover all the interactions that occur in the reasoning space. It focuses only on the interaction between different variables and ignores the interaction between the values of the same variable. Such interactions are the consequence of temporality. As the time passes, the values of variables change, and these changes affect the reasoning process. I proposed multi-dimensional covering arrays to address this *dimension*. MDCA extends MCA by including intra-variables interactions into consideration. Therefore, MDCA handles interactions between variables values and between values of the same variable.

MDCA and MCA noticeably reduce the number of data needed to test the system. Furthermore, since MDCA covers both intra and inter-variable interactions, it comprises data sets that are longer than MCA. Empirical evidence reveals that the length of data sets is well rewarded in terms of conditions coverage. MDCA covers more conditions than MCA. Still, such coverage can be biased by the length of the data set and be the result of a random process. The empirical evidence disproves this hypothesis. The results obtained using randomly generated data perform poorly in comparison to MDCA, therefore, it is not the length but the properties it withhold that makes it cover more conditions. An important

lesson learned from the empirical evidence, is that MCA performs better in systems that ignore history. In such cases, MCA provides shorter data sets with the same results that MCA.

Constructing MDCA is not trivial. I provide algorithms and implementations that are a proof of the concept and show the feasibility of constructing MDCAs. They leave room for many optimizations in terms of performance. The optimization problem itself leaves room for optimization. One possible optimization could be including the length of the array, however, this will introduce new problems such as defining a new expected value (the expected value now may vary according to the sequence length) and termination criterion.

Finally, the experimental assessment I conducted is far from being perfect. Like any other experiment it is exposed to bias introduced by the algorithm implementation, the experimental subjects, and the evaluation criteria. Despite these threats, I think that it provides valuable insight information about how MDCA would perform at finding faults because: (i) *the construction algorithms produce data sets that satisfy all the MDCA properties*; (ii) *my hypothesis is that the more the conditions covered, the higher the probability of finding faults*; and (ii) *the experimental subjects contain fairly complex decision making*.

Chapter 4

Specifying aspect-oriented adaptation mechanism

Aspect-oriented programming (AOP) enables developers to introduce structural and behavioral modifications into a system without requiring them to modify the system's code. Developers can then introduce, augment, or replace existing behavior and structure by weaving *aspects* into the system.

Aspects are the base elements of AOP, they consist of *point-cut descriptors*, *advices*, and *inter-types*. Point-cuts designate well-defined points in the system code, where advices are woven. Advices are modules that encapsulate the behavior that is injected into the system, whereas inter-type are elements and modifications that apply over the system structure. When woven, aspects (1) inject the advices in the points designated by point-cuts, and (2) perform the structural modifications described by inter-types.

AOP has good characteristics to make an adaptation mechanism. It can introduce / remove behavior, modify / rollback structure at compile and run time. Adaptations can be achieved by weaving or unweaving aspects, which will introduce the structural and behavioral modification that will adapt the system. Then, developers can implement adaptation concerns such as encryption in Cordelia's ACRM system using aspects (cf. Section 2.1.3).

Nevertheless, the same mechanisms that make AOP a good adaptation mechanism also introduce interference, invasiveness, and evolution problems. These problems threat the validation of aspect-oriented programs, and hamper the adoption of AOP [166]. Consequence of these problems developers need to commit much effort and time finding faults accidentally / obliviously introduced by aspects. In adaptation, these problems may cause the system to break down when new aspects are woven into the system.

In this chapter I propose a specification framework based on a characterization of AOP invasiveness [165]. First, I briefly introduce AOP and AspectJ (the *de facto* standard for AOP) to later present a motivating case study that describes AOP's problems and their impact on validation. Next, I present the ABIS specification framework and the ABIS tooling

support to later show how it help developers reducing the validation overhead and increase AOP's confidence.

4.1 A brief introduction to AOP and AspectJ

In aspect-oriented programming (AOP), aspects are defined in terms of two units: *advices*, and *point-cut* descriptors (PCD). Advices are units that realize the crosscutting behavior, and point-cuts are pointing elements that designate well-defined points in the program execution or structure (*join-points*) where the crosscutting behavior is executed. I illustrate these elements through two code fragments belonging to a banking aspect-oriented application. The first (listing 1) presents the PCD declaration for logging (lines 2-5) and transaction (lines 7-10) concern, whereas the second (listing 2) presents an advice (lines 3-14) realizing a transaction concern.

```
1  public aspect BankAspect {
2      pointcut logTrans(int amount):
3              ( call(boolean Account.withdraw(int)) ||
4                call(boolean Account.deposit(int))
5              ) && args(amount);
6
7      pointcut transaction(): execution(boolean Account.*(int))
8                          && cflow(execution(void Bank.operation(..))
9  }
```

Listing 4.1: Aspect with two pointcuts

In AspectJ, a PCD is defined as a combination of *names* and *terms*. Names are used to match specific places in the base system and typically correspond to a method's qualified signature. For instance, the name boolean `Account.withdraw(int)` in listing 4.1 (line 3) matches a method named `withdraw` that returns a type `boolean`, receives a single argument of type `int`, and is declared in the class `Account`.

Terms are used to complete names and define in which conditions the places matched by names should be intercepted. AspectJ defines three types of terms: wildcards, logic operators, and keywords. The combination of names and terms is referred as expression.

Wildcards serve to enlarge the number of matches produced by a name. The AspectJ PCD language defines two wildcards: "*" and "..". Logic operators serve to compose two expressions into a single expression, or to change the logic value of an expression. The AspectJ PCD language provides three logic operators, "&&" (conjunction), "||" (disjunction), and "!" (negation).

Keywords define when and in which conditions the places matched by names should be intercepted. The AspectJ PCD language defines 17 keywords for that purpose. For instances, the keyword `call` in the logTrans PCD (lines 3, 4) indicates the interception of all the calls to the enclosed names, whereas the keyword `args` (line 5) indicates that the PCD argument amount should be the argument of those invocations.

Some keywords point to joint-points that can be computed only at runtime. The AspectJ PCD language defines 6 keywords for that purpose: `cflow`, `cflowbelow`, `if`, `arg`, `this`, and `target`. The transaction PCD (lines 7-10) incorporates this kind of keywords. It contains two expressions: (1) a static expression that intercepts the execution of any method returning a boolean in the class Account (line 8); (2) a dynamic expression that constrains the interception of the static expression to the execution occurring inside the control flow of the execution of the method operation in the class `Bank`. This is a dynamic expression since determining whether the execution of a method occurs during the execution of another can be done only at runtime. I refer to join-points occurring only at runtime as dynamic join-points and PCDs pointing these points as dynamic-PCDs.

AspectJ extends the Java syntax to allow developers to implement advices as natural as possible. Advices can be seen as routines that are executed at some point. Typically AspectJ advices are bounded to a PCD designating the points where they will be executed. For instance, the advice in listing 2 (lines 3-14) is bounded to the PCD transaction (line 3). AspectJ provides three different kinds of advices `before`, `after`, and `around` indicating the moment when they are executed. Before and after indicate that the advice instructions are executed respectively before and after the intercepted method. Around indicates that the advice can perform computations before, after, or instead the intercepted method (the advice embodies the call to the intercepted method).

```
1  public aspect ModifyAccount {
2      public interface GeneralAccount{
3          public void CalculateDebt();
4      };
5
6      declare parents : Account implements GeneralAccount;
7
8      public Double Account.debt = new Double();
9
10     public void Account.calculateDebt(){
11         ...
12     }
13 }
```

Listing 4.2: Aspect modifying the structure of the base system

AspectJ aspect can also introduce structural classes' attributes, interfaces, and methods, and modify the class hierarchy of the base system. Listing 4.2, present an aspect that introduces a new interface `GeneralAccount` (line 2), modifies the class `Account` by changing its hierarchy (line 6), and introduces a new method `calculateDebt` (line 8) and field `debt` (line 10) into the class `Account`.

4.2 Motivating Case Study

In this section, I illustrate AOP's problems when multiple aspects are introduced into the existing system (base system + aspects). My goal is to show that aspects offer efficient mechanisms to implement crosscutting (and adaptive) concerns, but that they can also introduce complex faults that are difficult to detect and trace back to their source.

To illustrate these issues, I present an example implemented in Java and AspectJ. The example is a chat application implemented with 5 aspects. I run system-level test cases on the application to validate the initial version. Then, maintainers evolve the application adding authentication capabilities. While testing the new version, faults are detected.

I precisely discuss the analysis performed to trace the source of the error back to a wrong interaction between aspects and the base system. Based on these observations, I motivate an approach to assist the validation of aspect-oriented programs.

Notice that though the case study is not a *self*-adaptive system, it reflects the possible faults that can occur when new aspects are introduced to reconfigure the system. The chat application comprises different aspects that can be woven / unwoven when needed. These aspects realize concerns that can be removed from the system if needed. That is, the system can be adapted by unweaving selected aspects.

4.2.1 A chat application

A chat application is a system that allows users to communicate with each other in real time. This chat is composed of two parts: a client and a server. The server handles client, manages the communication between them, and ensures their uniqueness. The client transmits messages that are dispatched to other clients through the server. The global behavior of this *chat application* can be described as follows. Initially, the server is waiting for clients. The establishment of communications is called association. The cease of communication is called disassociation. Before associating a client, the server checks the uniqueness of the client's nickname. Clients using existing nicknames are not associated. Once associated, the clients can send messages. Such messages are encrypted and decrypted by the clients. Each client's chat session is recorded in a log file. The server also stores each session into a log file. A graphical user interface (GUI) controls the client and server behavior.

4.2.2 Initial version

Developers have implemented the chat application in Java ™. From the requirements documents, developers have separated the core concerns form the crosscutting (and adaptive) concerns. Figure 4.1 shows the class diagram for the core concerns of the chat application.

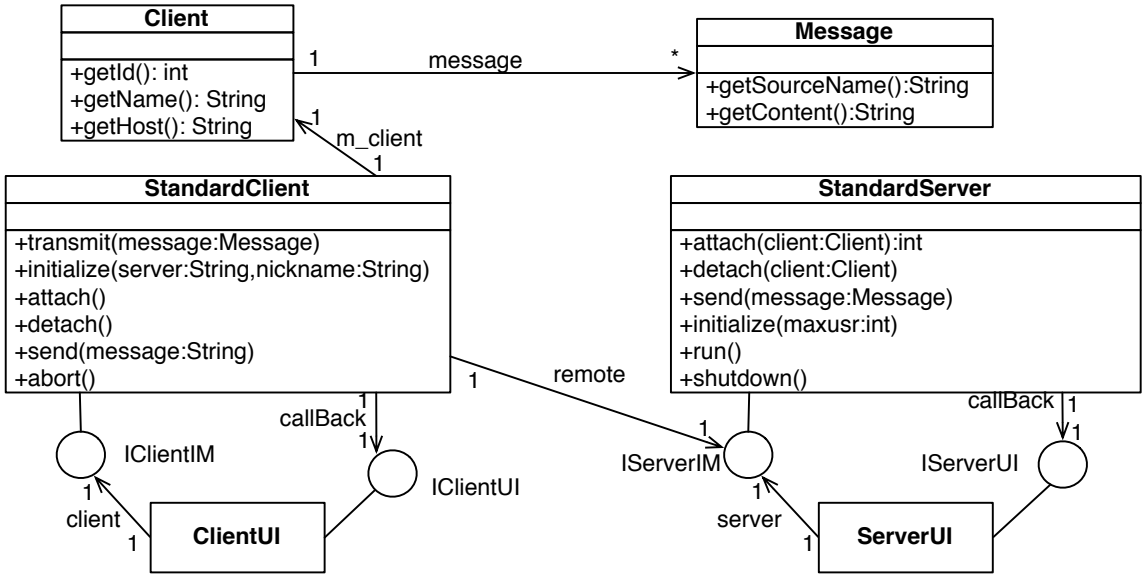

Figure 4.1: *Chat application* class diagram

`IClientIM` and `IServerIM` are the remote interfaces for the client and the server. The `StandardClient` class realizes the client interface. The methods `attach()` and `detach()` associate and dissociate the client to a server. The method `transmit(Message)` notifies the GUI about the arrival of a new message. Finally the method `abort()` aborts the execution of the client. The `StandardServer` class realizes the server interface. As in the client, the methods `attach(Client)` and `detach(Client)` associate and dissociate a client. The `send(Message)` method dispatches the messages to other clients associated to the server. The class `Client` is a container for the client's information. The class `Message` supports the messaging mechanism between client and server. Client and Server communicate using the Java RMI distribution mechanism [209]. The client and the server GUI are implemented with the Standard Widget Toolkit (SWT).

4.2.3 Crosscutting concerns

Developers have identified and implemented with AspectJ the following crosscutting concerns. Notice that these concerns are independent from each other and can be woven

/ unwoven from the chat application at will.

- ***Message encryption*** encrypts and decrypts the incoming / outgoing messages. Encryption / decryption occurs just before the execution of the methods `send` and `transmit`. It replaces the method arguments with the encrypted / decrypted version of the message.
- ***Message logging*** logs the incoming / outgoing messages for each user. It executes before transmitting a message and after receiving a message (before the encryption and after the decryption).
- ***Error handling*** captures the communication exceptions and raises an alert indicating communication problems. It executes after an exception of type remote is thrown.
- ***Server logging*** logs the server activity in a file. It executes before and after all the methods of the `IServerIM` interface. It observes the execution of each server's method.
- ***Nickname uniqueness*** checks the existence of only one nickname in the server. It executes just before the server's association methods (`attach` and `detach`). In the case of association, it checks the existence of the client's nickname on the server. If the nickname exists, it throws an exception. Otherwise, it adds the name to a list and executes the association normally. In the case of dissociation, it removes the client's nickname from a list.

```
1  public aspect EncryptionAspect{
2      pointcut encryptMessage(String string):
3          execution(void IClientIM.send(String)) && args(string);
4
5      pointcut reencryptMessage(Message messg):
6          execution(void *.retransmit(Message)) && args(messg);
7
8      pointcut decryptMessage(Message messg):
9          execution(void IClientIM.transmit(Message)) && args(messg);
10
11     void around(String arg) : encryptMessage(arg){
12         arg = encrypt(arg);
13         proceed(arg);
14     }
15
16     void around(Message message) : reencryptMessage(message){
17         message.sContents=encrypt(message.sContents);
18         proceed(message);
19     }
20
```

```
21      void around(Message message) : decryptMessage(message){
22          message.sContents=decrypt(message.sContents);
23          proceed(message);
24      }
25  }
```

Listing 4.3: "Implementation of the encryption concern"

Listing 4.3 shows the implementation of the *Message encryption* concern. It is clear from the code that the three advices of this aspect are changing the argument values of the intercepted methods (lines 11-14, 16-19, 21-24). The original message is replaced by the encrypted / decrypted version and then it is re-injected to the original method (`proceed` call). This concern can only be implemented by using invasive aspects, otherwise it must be hard-coded in the base system. It also a typical example of an adaptive concern, which can be be used to adapt the system. Different network requirements may require or not encryption. In unprotected networks the encryption aspect will be woven, whereas in secure networks in will be unwoven. Furthermore, there exist different flavors of encryption that can be used. Each of these flavors (or encryption algorithms) can be encoded into an aspect.

```
1   public aspect UniqueNameAspect issingleton() {
2       public pointcut ensureUniqueness(IServerIM serv,Client client):
3           execution(* IServerIM.attach(Client)) && target(serv) && args(client);
4
5       public pointcut removeFromList(IServerIM serv,Client client):
6           execution(* IServerIM.detach(Client)) && target(serv) && args(client);
7
8       private static ArrayList nameList=new ArrayList();
9
10      int around(IServerIM serv,Client client)
11          throws UsedNameException:ensureUniqueness(serv,client){
12          int retValue=-1;
13          if(!nameList.contains(client.getSName())){
14              nameList.add(client.getSName());
15              retValue=proceed(serv,client);
16          }
17          else{
18              throw new UsedNameException();
19          }
20          return retValue;
```

```
21      }
22
23      after(IServerIM serv,Client client): removeFromList(serv,client) {
24          nameList.remove(client.getSName());
25      }
26  }
```

Listing 4.4: Implementation of the unique nickname concern

Listing 4.4 shows the implementation of the *Nickname uniqueness* concern. This aspect manages a list of the currently associated clients nicknames (`nameList` line 8). If the actual client nickname does not exist in the list (line 13) then it is added to the list (line 14) and the intercepted method is executed. Otherwise, the method is never executed and an exception is raised. This aspect conditionally replaces the execution of the methods it intercepts. Analogously to the *Message encryption* concern, it can only be implemented by using invasive aspects. This concern also exemplifies an adaptive concern. Particular users may want their server to be able to accept clients with existing nicknames. In this case, the nickname uniqueness aspects can be unwoven from the base system.

These examples illustrate how aspects help implementing the crosscutting and adaptive concerns that modify the flow and / or data in the base system.

4.2.4 Validating the initial version

I test the initial version with 7 system-level test scenarios. Each scenario validates a different dimension of the system. These dimensions are summarized as follows:

1. The association mechanism between client and server.
2. The association mechanism supports multiple clients.
3. Clients can send/receive messages.
4. The server distributes the messages among clients.
5. *The server detects clients named with an existing nickname.
6. *The server association mechanism removes a used nickname when disassociating a client.
7. *The error handling mechanism handles the exceptions.

All these test scenarios pass on the initial version composed of the core concern and 5 aspects. Test scenarios marked with "*" were explicitly designed to test the functionalities introduced by the aspects. The other test scenarios were designed to test general system functionalities.

4.2.5 Evolving the chat application

The initial version of the chat application allows any user to associate with a server. Here, I add an authentication mechanism to ensure that only registered users are able to associate with a server. We also want to ensure that the clients of this new version are compatible with the old version. As a consequence, a server without authentication must be able to associate an authenticated client. Authenticated servers must refuse the association of unauthenticated clients. The chosen authentication protocol proceeds as follows: the client provides the nickname and password data to the server. The server checks the nickname, password pair internally. If the pair is authentic, then the server will associate the client, otherwise the client is not associated.

The maintainers have implemented this evolution by adding two classes to the original design: `AuthenticatedSever` that extends `StandardServer` and overrides the method `attach(Client)`; `AuthenticatedClient` that extends `StandardClient` and overrides the methods `transmit(Message)` and `receive(Message)`. Additionally, they have performed minor changes in the class `Client` and the client GUI adding support for password input. Since the evolution augments the old behavior, the crosscutting concerns remain unchanged. Notice that these modifications can also be introduced using aspects. This will allow to adapt the chat server / client to support authentication concerns. Appendix B, presents the code for aspects introducing the structural modifications I previously described.

Although maintainers were aware of the presence of aspects, they were not able to predict the potential side effects of aspects. To do so, maintainers must be capable of performing global reasoning. That is, reason about each part of the system in such a way that all the interactions are known. Sadly, it is no always possible to globally reason about the system. As the system grows in size and complexity, global reasoning becomes more difficult. Therefore, maintainers tend to make the assumption that if the evolved system does not change its old behavior, aspects will behave as expected. Moreover, the obliviousness property of AOP [80] claims that maintainers should be unaware of aspects implementing crosscutting concerns. That is, maintainers do not have to take care about the aspects.

In *self*-adaptive systems, where aspects realize adaptive concerns, it is even more difficult to perform global reasoning and foresee all the interactions between aspects and aspect with the base system. *Self*-adaptive systems are constantly mutating and producing new interactions between aspects. As new aspects are woven / unwoven, unforeseen interactions occur.

4.2.6 Validating the new version

I use the previous test scenarios for regression testing. To do so, I replace the standard client/server with the authenticated one. I also add 5 scenarios to validate the authenti-

cation mechanism as well as the compatibility between the two versions. The dimensions addressed by these scenarios are summarized as follows:

8. The authentication mechanism detects invalid nicknames and passwords.
9. The server detects void nicknames or password (or both).
10. The client association mechanism is compatible with the standard server.
11. The authenticated server is incompatible with the standard client.
12. The authenticated client messaging mechanism is able to send messages to standard clients.

The results of the executing tests 1 to 12 are summarized in table 4.1.

Table 4.1: Test results after evolution. ✓*indicates that a test passes, X indicates that a test fails.*

Test	1	2	3	4	5	6	7	8	9	10	11	12
	x	x	x	x	x	x	x	✓	✓	✓	✓	x

4.2.7 Reasoning about the problems

Looking at the results of the test scenario execution it is not easy to see where the problems are located. However, a rigorous manual analysis will help detecting the problems. We will deal with the authenticated association issues and later the compatibility issues.

The failure of the two first test cases tells that the client cannot associate with the server or the server cannot authenticate / associate the clients. The failure of tests 3, 4, 5, 6 and 7 are consequence of the failure of the test case 1 and 2. This is because the preconditions cannot be fulfilled: there is no connected client. The success of tests 8 and 9 provides no information about the association mechanism. The success of the first test is fundamental to obtain more information about tests 3 to 7. This guides our analysis to examine the association and the authentication mechanism.

After rigorously inspecting the base system we found no errors, however, we need to take into account that the chat runs with aspects. Before examining the aspects code searching for errors, we try to run the server without aspects that could interfere with the association/authentication mechanism.

We remove only the invasive aspect *Nickname uniqueness* because it is the only one capable of interfering with the association/authentication mechanism. Moreover, the test scenarios reporting errors evaluate only the base concerns implemented in the base system (except for the test case 5, which test a functionality introduced by aspects, and is expected to fail). Therefore, if the test cases pass without the aspect, then the problems are due to wrong interactions between the aspect and the base system.

Table 4.2 summarizes the test results after removing the *Nickname uniqueness* aspect. Now test cases 1 to 4 pass, however, the test case 5 fails because it depends on the aspect.

Table 4.2: Test results after removing the *Nickname uniqueness* aspect

Test	1	2	3	4	5	6	7	8	9	10	11	12
	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	x

Removing the aspect helped to localize the problem to a bad interaction (the *Nickname uniqueness* aspect is interfering with the association mechanism). Nevertheless, we still ignore the specific localization of the failure and its cause.

After rigorously inspecting the aspect code and the places it affects, we finally localize the specific cause of the problem. The problem occurs because the advice (listing 4.4 lines 10-21) declared on the aspect captures a wrong join point, the execution of the `attach` method in the `StandardServer` class. The point-cut descriptor (listing 4.4 lines 2-3) used by the advice captures all the executions of the `attach` method in the server. This, while the body of the advice realizing the crosscutting concern is designed to be executed just once at each join point. This means, once from the beginning to the end of the method execution.

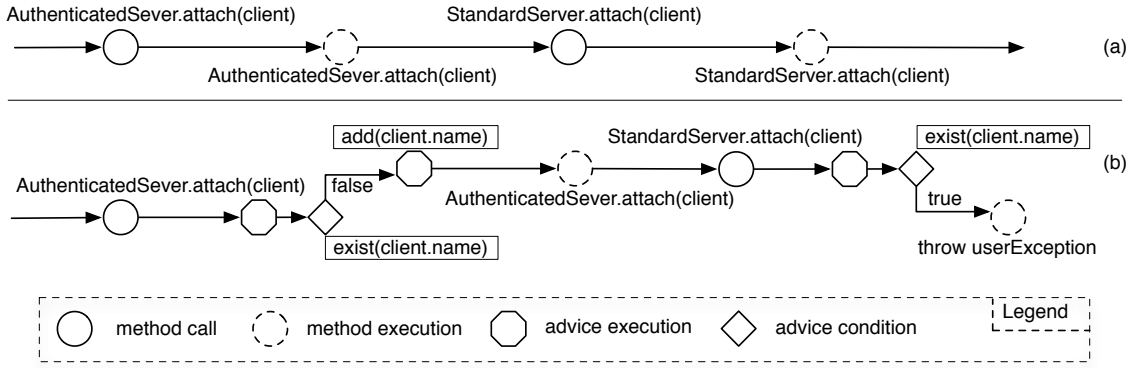

Figure 4.2: Schematic view of the association flow. (a) association without the *Nickname uniqueness* aspect. (b) association with the *Nickname uniqueness* aspect.

Figure 4.2 depicts schematically the association flow without (a) and with (b) the *Nickname uniqueness* aspect. When woven (b), the aspect captures the first call performing the authentication (`AuthenticationServer.attach(client)`). Then, the aspect finds that the client nickname is not in the registry (`exist(client.name)=false`) and adds it into the registry (`add(client.name)`). The aspect also captures the second call that performs the association (`StandardServer.attach(client)`). It is here where the aspect finds that the name exists in the registry and throws an exception. This problem is hard to detect because it can be a combination of a failure in the point-cut descriptor and a limited implementation of the advice (the advice is designed to advise only once the join-point). To solve this

problem we could modify the advice implementation or the point-cut descriptor. However, I think that the advice implementation realizes in a proper manner the crosscutting concern. Therefore, I modify the point-cut description. I also modify the point-cut descriptor used by the disassociation advice (listing 4.4 line 19-21) to be executed just once after disassociating a client.

```
1  public pointcut ensureUniqueness(IServerIM remote,Client client):
2      execution(* AuthenticatedServer.attach(Client)) &&
3      target(remote)&&args(client);
4  public pointcut removeFromList(IServerIM serv,Client client):
5      execution(* AuthenticatedServer.detach(Client)) &&
6      target(serv) && args(client);
```

Listing 4.5: Modified point-cut descriptor of the*Nickname uniqueness* aspect

Listing 4.5 presents the code of the modified point-cuts of the *Nickname uniqueness* aspect. The new descriptor targets the `AuthenticatedServer` class instead of the `IServerIM` interface. This ensures that only one call to the `attach` method will be captured.

Once I have detected and fixed the association problem, I explore the compatibility problems. The failure of test 12 is due to a compatibility problem when sending / receiving messages. Following the procedure we used to localize cause of the association issue we localize the case of the compatibility issue. The problem is localized in the *Encryption* aspect and is analogous to the association problem. The advice captures the execution of the messaging methods twice, therefore, it encrypts / decrypts the messages twice. On the other hand, the standard client encrypts / decrypts the messages only once. Analogously to the previous problem, we solved this problem by changing the point-cut descriptor to match only one execution each time. That is, the new descriptor targets the `AuthenticatedClient` class instead of the `IClientIM` interface.

4.2.8 Discussion

Through this experiment I have shown that despite the features provided by aspects, they can hamper the software evolution and variability through aspects. The main issue is that it is very hard to reason about the aspects impact on the final system, and it is very hard to trace the source of the problems to aspects.

The successive execution of a set of test scenarios and the manual inspection of code helped us tracing the problems to aspects. Nevertheless, this process is tedious, time consuming and error prone. Besides, there is no generic formula to trace this kind of problems. In the chat application, the removal of aspects helped the developers to localize the problems. This was possible because the test scenarios we used were testing the base system's

functionalities. Nevertheless, it is not always possible to simply remove the aspects. In the case of *self*-adaptive systems, aspects are meant to be woven / unwoven, however, there may be so many aspects, that trying the combinations of aspects that may be interfering / interacting may not always be possible.

Nowadays there is a missing element in the AOP support. The tedious process I have performed tells us that there is a need to abstract from code to reason about the interactions between aspects and the base system. To tackle this issue, I propose a framework for specifying (i) *the invasiveness patterns that aspects realize*, and (ii) *the invasiveness patterns expected on the base system*. This means specifying the interaction between aspects and base system. This framework also checks whether the pattern realized by aspects do not violate the set of expected patterns declared on the base system. These specifications will explicitly state the intentions of the aspects with the base program and with other aspect. This will assist developers to localize and solve problems due to faulty invasive aspects.

4.3 Specifying aspects-base system interaction

The specification of interactions between aspects and base system consists of two parts. In the first, developers characterize aspects with specific invasiveness patterns (*Aspect specifications*). In the second, developers specify the invasiveness patterns that the base system allows from aspects.

For the first, I propose to characterize the different invasiveness patterns that aspects can realize [164]. For the second, I propose to specify assertions that allow or forbid invasiveness patterns to interact with the base system's elements.

The goal of these specifications is to obtain information about the potential unexpected interactions that invasive aspects can produce. Such information will help developers to reason about the harmfulness of aspects. Therefore, it assists developers to track potential faults (introduced by invasive aspects).

4.3.1 Aspect specification

In [164] I present a classification of invasive aspects. Such classification is the result of an analysis of the invasive mechanisms that AspectJ [227] provides. It allows me to identify specific invasiveness patterns and therefore abstract from code. Such abstraction helps developers to reason about the interaction of aspects and the base system. I use this classification to characterize aspects with invasiveness patterns.

AspectJ [227] is the most prominent realization of invasive aspects. It realizes the crosscutting behavior on *advices* and designating the places where this behavior must be woven with point-cut expressions. Point-cuts are regular expressions that match well-defined points in the structure (static) and execution (dynamic) of a Java™program. In AspectJ, aspects are composed of advices, point-cuts, and inter-type declarations. Advices

are used to modify the program flow and write or read fields. Advices can be declared inside a privileged aspect, which means that they are enabled to ignore the object-oriented access policy. Inter-type declarations are used to introduce methods and fields into a target class. Inter-type parent declarations are used to modify the class hierarchy.

Developers can modify the program structure at the aspects level (using an aspect), whereas behavior is modified at the advice level (through an advice). In the following, I list the classification elements with a brief description. Aspects invasiveness patterns are marked with △, and advice invasiveness patterns are marked with ◊.

◊ *Augmentation*: After crosscutting, the body of the intercepted method is always executed. The advice augments the behavior of the method it crosscuts with new behavior that does not interfere with the original behavior. Examples of this kind of advices are those realizing logging, monitoring, traceability, etc.

◊ *Replacement*: After crosscutting, the body of the intercepted method is never executed. The advice completely replaces the behavior of the method it crosscuts with new behavior. This kind of advices eliminate a part of the base system.

◊ *Conditional replacement*: After crosscutting, the body of the intercepted method is not always executed. The advice conditionally invokes the body of the method and potentially replaces its behavior with new behavior. Examples of this kind of advices are advices realizing transaction, access control, etc.

◊ *Multiple*: After crosscutting, the body of the intercepted method could be executed more than once. The advice invokes two or more time the body of the method it crosscuts generating potentially new behavior.

◊ *Crossing*: After crosscutting, the advice invokes the body of a method (or several methods) that it does not intercepts. The advice have a dependency to the class owning the invoked method(s).

◊ *Write*: After crosscutting, the advice writes an object field. This access can break the protection declared for the field and can modify the behavior of the underlying computation.

◊ *Read*: After crosscutting, the advice reads an object field. This access can break the protection declared for the field and can potentially expose sensitive data.

◊ *Argument passing*: After crosscutting, the advice modifies the argument values of the method it crosscuts and then invokes the body of the method. The body of the method always executes at least once.

△ *Hierarchy*: The aspect modifies the declared class hierarchy. For example, the aspect adds a new parent interface to an existing one.

△ *Field addition*: The aspect adds new fields to an existing class declaration. These fields depending on their protection can be acceded by referencing an object instance of the affected class.

△ *Operation addition*: The aspect adds new methods to an exiting class declaration. These methods depending on their protection can be acceded by referencing an object in-

stance of the affected class.

All the advices of the *Encryption* aspect (Listing 4.3) are classified *Argument passing* and two of them *Read* and *Write*. That is because they modify the argument values of the methods they intercept and two of them read and write the value of the field `sContents` (lines 18,22). The advices of the *Nickname uniqueness* aspect (Listing 4.4) are classified *Conditional Replacement* (lines 10-21) and *Augmentation* (lines 23-25). The first conditionally replaces the execution of the intercepted methods and the second is orthogonal to the intercepted methods.

It is worth mentioning that the patterns *Augmentation*, *Replacement*, *Conditional replacement* and *Multiple* are exclusive. An advice can be classified with only one of them.

4.3.2 Core specification

Developers may specify the base system (core) by asserting the invasiveness patterns allowed or forbidden to interact with it. Such specification comes from the base system designers and declares an expected interaction.

Aspects crosscut the base system at the level of classes (modifying the class structure), methods (modifying the declared behavior) and fields (accessing the data contained in object fields). Therefore, specifications are attached to each one of these elements. By default (implicit specification) only the patterns *Augmentation*, *Crossing*, *Read* and *Field addition* are allowed to advise the base system. This is because a priori these classes do not alter the program flow, data or structure; hence, they are less harmful than the others.

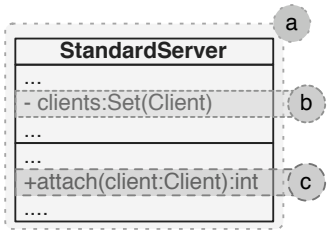

Figure 4.3: base system specification covering

Specification for classes: Specify the invasiveness patterns allowed at two levels. The first is related to the class structure modifications such as the addition of a field. The second is related with methods and fields declared in the class. A class is specified with an allowed invasiveness patterns that applies to its fields and methods. This specification can allow or forbid any invasiveness pattern forbidden or allowed by default. It corresponds to (a) in figure 4.3 covering all the class definition.

Specification for fields: Specify the invasiveness patterns that a field allows in terms of how advices access it. This specification can allow *Write* and forbid *Read*. It corresponds to (b) in figure 4.3 covering only a field definition.

Specification for methods: Specify the invasiveness patterns allowed on a specific method. This specification can allow *Replacement*, *Conditional replacement*, *Multiple*, *Write*, *Argument passing* and forbid *Augmentation*, *Crossing*, *Read*. It corresponds to (c) in figure 4.3 covering only a method definition.

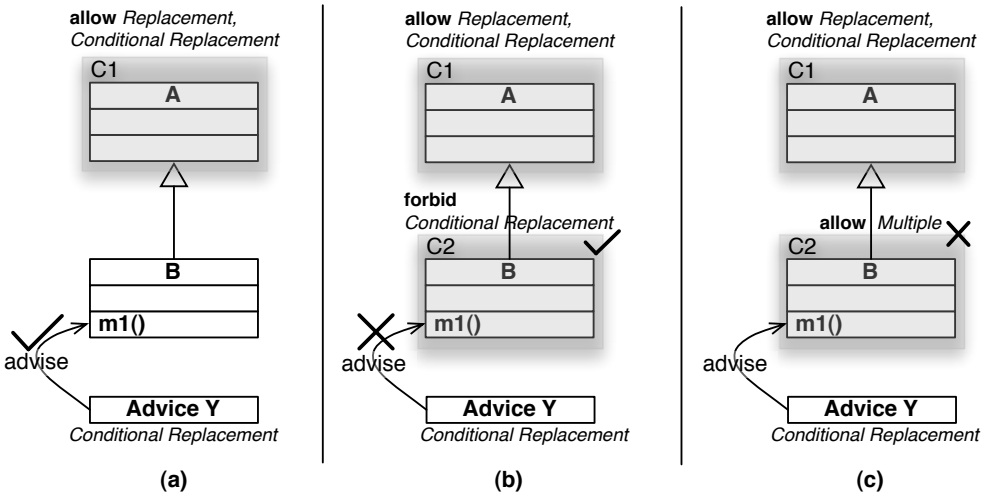

Figure 4.4: Example of specification reuse

A class specification can be reused in the case of the inheritance when top-level specifications are inherited by leaf classes. Let the class `B` extends class `A`. If `C1` is the specification of `A`, and `B` is not explicitly specified, then `C` is the specification of `B`. Furthermore, if `C1` allows a set of patterns `S`, then a new specification of `B` can only forbid a pattern in `S` but not allow other patterns. Figure 4.4 depicts an example of specification reuse. In the figure, a class `B` with a method `m1()` extends a class `A`. The specification `C1` of `A` allows the patterns *Replacement* and *Conditional Replacement* to advise the class and its methods, fields. The advice `Y` realizes the invasive pattern *Conditional Replacement* and advises the `m1()`. In (a), `B` reuses the specification of `A`, and therefore, the advise `Y` is allowed advise `m1()`. In (b), `B` adds its own specification forbidding the pattern *Conditional Replacement*, and since the specification is valid, the advise `Y` is forbidden to advise `m1()`. Finally, in (c), `B` adds its own specification allowing the pattern *Multiple*, however, the specification is invalid because `B` cannot allow patterns forbidden in `A`.

In the case of conflicts between the specifications of fields, methods and classes I propose the following mechanism: specifications of fields are always used if they exist. If a method is specified (explicitly), its specifications are used instead of the global ones. Figure 4.5, depicts an example of the conflict resolution. In the figure, a class `A` with a field `f1` and a class `D` with a method `m2()`. The specifications `C1` of `A` and `C2` of `D` allow respectively the patterns *Write* and *Multiple*. The advice `Y` realizes the invasive pattern *Replacement*

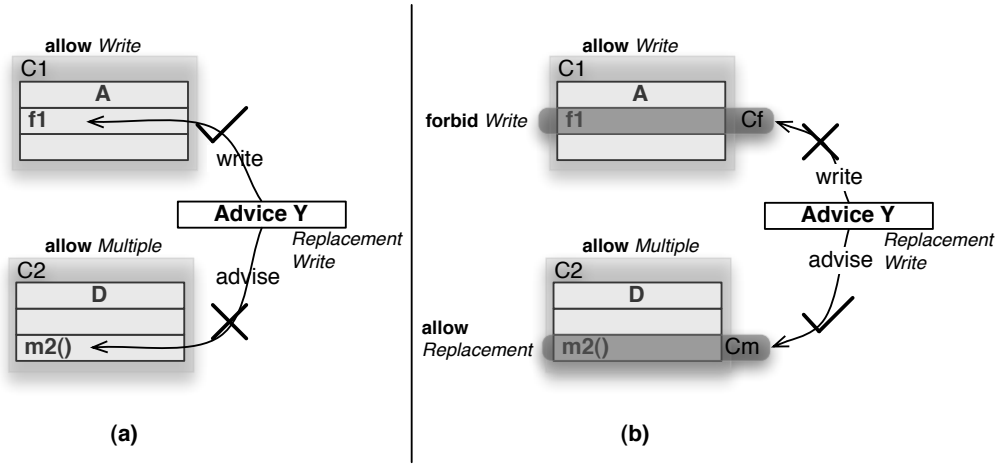

Figure 4.5: Example of conflict resolution

and *Write*, advises the method`m2()` and writes the field `f1`. In (a), the specification `C1` is applied on `f1`, and therefore `Y` is allowed to write it. The specification `C2` is applied on `m2()`, and therefore, `Y` is forbidden to advise it . In (b), `f1` and `m2()` add their own specification forbidding the pattern `Write` and allowing the pattern `Replacement` respectively. These specification override `C1` and `C2`, therefore, `Y` is allowed to write `f1` but is forbidden to advise `m2()`.

Core specification formal interpretation

To define precisely how we interpret the base specification defined by a programmer on a base program, we use Alloy [115]. In this specification, we define the essential concepts of our base program: (`Class`, `Operation`, `Property`) that inherit from `SpecElement` (see Listing 4.6). A Class can have some properties, some operations and some super-classes. A SpecElement has some specifications that define the allowed and the forbidden types of advice declared by the programmer (`forbiddenAdviceType`, `allowedAdviceType`) on this base element and some specifications that define the allowed and the forbidden advice type really applied on this base element (`allForbiddenAdviceType`, `allAllowedAdviceType`). The alloy specification defines precisely how we compute the allowed and forbidden types of advice that can be applied on a base element depending of the programmer specification. The list of available types of advice is based on the classification presented in the previous sub-section. The formal interpretation of the specification propagation is defined by a set of alloy facts. Six assertions check that the propagation of specification keeps the consistency of the base program specification. The whole specification is provided in appendix C.

```
1  abstract sig SpecElement extends NameElement {
2      forbiddenAdviceType,allowedAdviceType : set AdviceType,
3      allForbiddenAdviceType,allAllowedAdviceType : set AdviceType,
4  }
5
6  sig Class extends SpecElement{
7      superclass : set Class,
8      properties : set Property,
9      operations : set Operation
10 }
11
12 sig Property,Operation extends SpecElement{}
13
14 abstract sig AdviceType {}
15
16 one sig Augmentation, Replacement,
17           ConditionalReplacement, Multiple,
18           Crossing,Write, Read,ArgumentPassing,
19           Hierarchy, FieldAddition,OperationAddition
20           extends AdviceType {}
21
22 fun getAllAdviceType() : set AdviceType{
23     Augmentation + Replacement + ConditionalReplacement
24     + Multiple + Crossing + Write + Read +
25     ArgumentPassing + Hierarchy + FieldAddition + OperationAddition
26 }
```

Listing 4.6: Core specification formal interpretation using Alloy

4.3.3 Specification matching

In order to detect when aspects or advices violate the core specifications it is necessary to check whether they match (agree) each other. An aspect or an advice violates a core specification when it realizes invasiveness patterns that the core specification forbid. This gives developers information about the invasive aspects harmfulness and assists them to reason about the impact of aspects on the composed system.

Violations of specifications are detected in the following way: st the aspect level, for each aspect we obtain the classes it targets adding fields, methods or modifying the hierarchy. Then, we compare the specification of forbidden patterns on each class with the

specification of the aspect. At the advice level, for each advice we obtain the methods it advises. Then, we compare the specification of forbidden patterns on each method with the specification of the advice. For the advices accessing fields, the matching is analogous to the previous.

Matching formal interpretation

The formal interpretation of the comparison between the core and the aspect specification is also specified using alloy. It defines the concept of `Aspect`, the concept of `Advice` that has a `type` and a set of `Joinpoints`. A fact defines that for all advice's joint-points, the type of advice is allowed by the base specification and not forbidden (see Listing 4.7).

```
1  sig Aspect extends NameElement{
2      advices : set Advice
3  }
4
5  sig Advice extends NameElement{
6      joinpointop : set SpecElement,
7      type : one AdviceType
8  }
9
10 fact adviceRight{
11     all a : Advice | all c : SpecElement |
12         a.type not in c.allForbiddenAdviceType and a.type in c.allAllowedAdviceType
13 }
14
15 assert AspectCanNotBe{
16     all a : Aspect | all c : SpecElement | all adv : Advice |
17         adv in a.advices
18         and #(c.allAllowedAdviceType )
19             - #(c.allAllowedAdviceType- adv.type) = 1
20 }
21
22 check AspectCanNotBe for 20
```

Listing 4.7: Aspect/Base formal interpretation using Alloy

4.3.4 Obliviousness

The obliviousness property of AOSD requires aspects to be transparent to the base code [80]. This means that the base code must be ignorant about aspects advising it. The placement of specifications between the aspect and the base system partially breaks the obliviousness property. Specifications make the base code aware about the existence of aspects that can potentially advise it. The obliviousness property can deteriorate a good design, and as exposed by Rashid and Moreira [192], abstraction, modularity, and composability are more fundamental properties.

Developers following the obliviousness property may ignore the existence of aspects or only take into account them when they know that they are modifying the points they attack. This may lead to an uncoordinated evolution of the aspect and the base system. Therefore, as the base system evolves and developers are oblivious, aspects can introduce side effects as we shown in section 4.2.7.

4.4 A specification framework for interactions

I have implemented a tool for matching specification as well a language to express the base system specifications called ABIS (Aspect-Base Interaction Specification) [1]. ABIS is built on top of the AJDT eclipse plug-in and is completely integrated with the eclipse IDE. After a short presentation of the global structure of ABIS I detail how each aspect and advice is automatically classified. Then, I will describe how to specify the base system with the expected invasiveness patterns.

Figure 4.6 presents the ABIS' organization. I have extended the AspectJ `AST Visitor` (1) in order to create a simplified (SAST) version of the Abstract Syntax Tree (AST). ABIS obtains information about the structure of the program (aspects and base system) from AJDT and builds a model of the program structure (2). This model contains the relations between aspects and the base system (advised and introduced element relations). An automatic classification process inspects the SAST and classifies each aspect and advice according to its invasiveness pattern (3). Then the model and the classified advices are checked following the previously presented matching process (4). If specification violations are found, then they are reported to the eclipse GUI (5).

4.4.1 Automatic classification of aspects

ABIS is able to automatically identify the invasiveness patterns I presented in section 4.3.1. It automatically inspects aspects and advices structure and classifies them according to their de-facto properties.

1. Publicly available at `http://contract4aj.gforge.inria.fr`

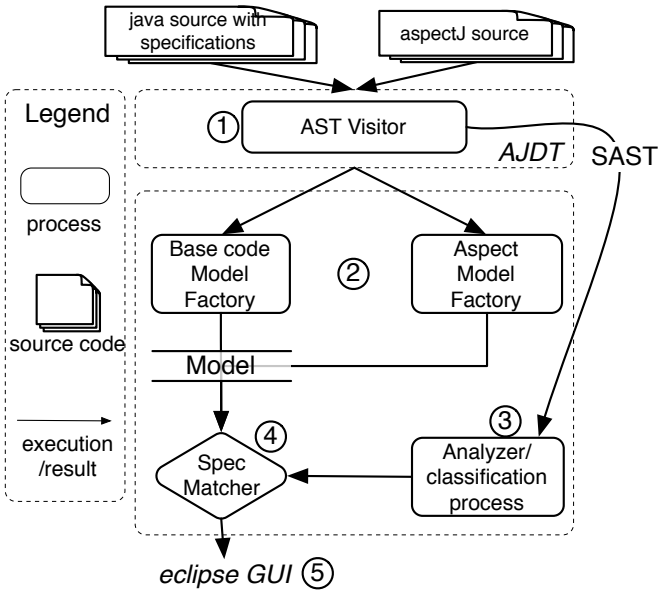

Figure 4.6: ABIS structure diagram

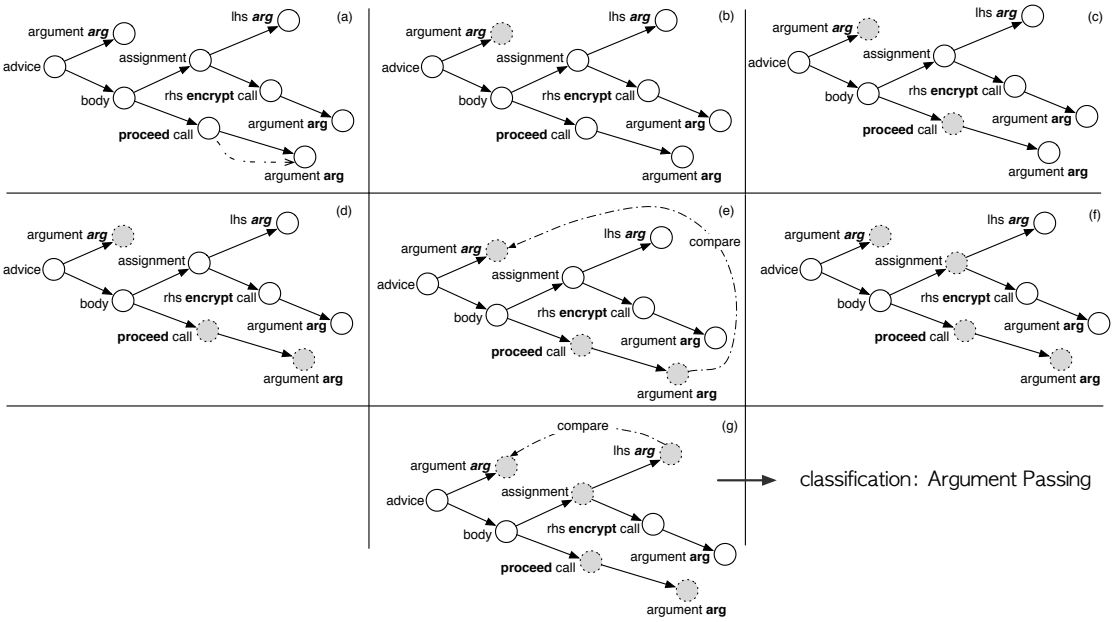

Figure 4.7: SAST and classification process of *Encryption* aspect, Listing 4.3, lines 9-12

Figure 4.7 (a), shows the SAST of the first advice of the *Encryption* aspect (Listing 4.3, lines 11-14). The root node of the SAST corresponds to the advice declaration. From the root node, we find the parameter declaration and the advice body node. The children of the body node are the statements declared on the advice. The node `assignment` corresponds to the assignment in line 12 of Listing 4.3. The node `proceed call` represents the call to the `proceed(arg)` method (line 13 of Listing 4.3) (it executes the intercepted method) and its children are the arguments it receives.

The classification algorithm applied to find the *argument passing* pattern in the *Encryption* aspect is the following (Figure 4.7 (b), (c), (d), (e), (f), (g)):

1. Initially, select all the advice argument nodes (Figure 4.7 (b)).
2. Traverse the SAST selecting all the nodes representing a call to the proceed method (proceed call node, Figure 4.7 (c)). Then, for each proceed call node:
   (a) Select all the proceed argument nodes (Figure 4.7 (d)).
   (b) If the selected argument nodes have different names than the advice argument nodes, then classify the advice as *Parameter passing* and stop this classification process. Otherwise continue (Figure 4.7 (e)).
   (c) Select all the assignment nodes on top of the current proceed call node (Figure 4.7 (f)). Then, for each assignment node:
      i. If the left hand side node (lhs) of an assignment has the same name as than one of the advice argument nodes, then classify the advice as *Parameter passing*. Stop this classification process (Figure 4.7 (g)).

This algorithm represents the set of rules used to identify the *Parameter passing* invasiveness pattern. In general, ABIS identifies invasiveness patterns by checking a set of rules to the advice SAST.

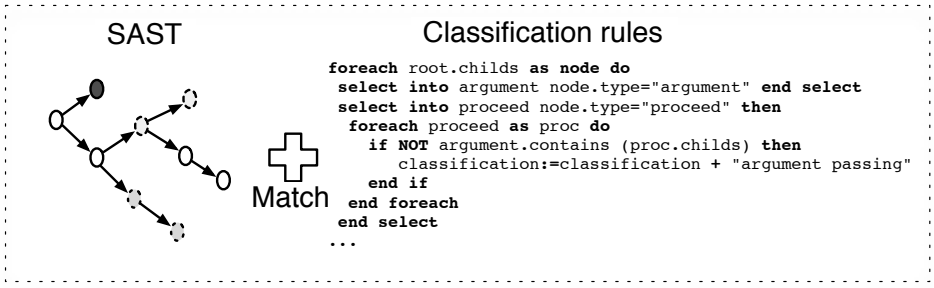

Figure 4.8: Automatic advice classification

Figure 4.8 illustrates the automatic classification process on advices. This process proceeds as follows: once obtained the advices SASTs a set of identification rules is applied

to them. The results of applying the rules are the invasiveness patterns realized by the inspected SAST. For instance, the steps presented in figure 4.7 (a)-(g) are the application of successive rules to detect the invasiveness patterns *Parameter passing*. Other rules can for example, check whether the left hand side of an assignment node is a reference to an object field, and then detected the invasiveness pattern *Write*.

Aspects invasive patterns are detected according to the aspect structural declarations. For example, if an aspect declares Inter-Type fields, then the detected invasive pattern is *Field Addition*.

4.4.2 Specifications in the base system

Base system specifications are expressed as meta-information over the code structure. Specifications take the form of Java 5 annotations [24]. The parameterized annotation `@Spec([allow=.. | forbid=..])` specifies the expected invasiveness pattern in the base system. It can be attached to classes, fields, and methods, and the possible values for its parameters vary as described in section 4.3.2.

The parameters `allow` and `forbid` are exclusive, i.e. only one can be used in each specification. The parameter `allow` indicates the invasiveness patterns allowed to interact with the base system. The default policy is to allow the non-invasiveness patterns *Augmentation*, *Crossing* and *Read*. Analogously the parameter `forbid` indicates the invasiveness patterns forbidden to interact with the base system.

```
1 @Spec(allow={"ConditionalReplacement"})
2 public synchronized int attach(Client client) throws RemoteException {
3   ...
4 }
```

Listing 4.8: `attach` method specified with an annotation

Listing 4.8 shows the `attach` method of the standard server specified with an annotation (line 2). This method allows advices realizing the following patterns to advise it: *Augmentation*, *Crossing*, *Conditional replacement* and *Read*. All the other patterns are forbidden.

4.4.3 Displaying violations

ABIS detects the aspects or advices violating the base system specification using the matching process presented in section 4.3.3. For each invasive pattern violating the core specification, ABIS raises a warning. Such warnings are displayed by using the Eclipse ™platform GUI. Figure 4.9, shows the warning messages displayed by ABIS when violations are de-

tected. The warning message contains references to the violator (aspect or advice) and to the element (class, field or method) to which is attached the violated specification.

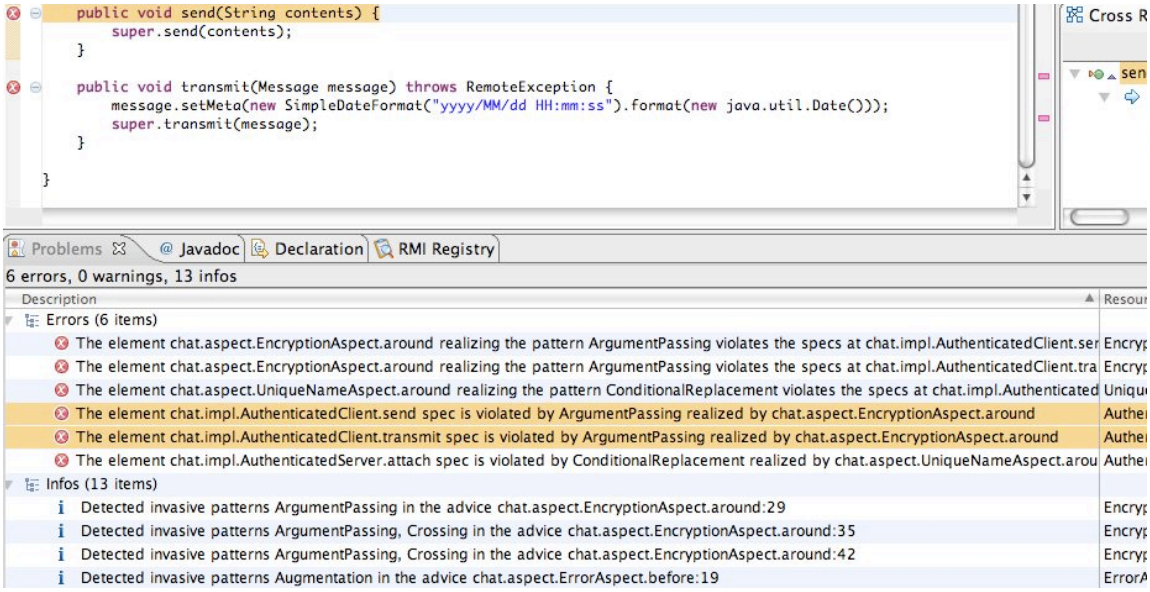

Figure 4.9: Warning raised by ABIS

These warnings enable developers to know the precise aspects that violate their specifications, and the precise base system's elements whose specifications are violated. The classification and matching process are executed at compile time, therefore, warning are displayed right after the code is compiled.

4.4.4 Contribution of the ABIS framework

ABIS statically computes and gives information, at compile-time about aspects violating base system's elements specifications. This information is useful and valuable:

1. Feedback for developers in the process of writing advices a specifying the base system.
2. For verifying an aspect-oriented program when aspects and the base system are developed separately.
3. For verifying an aspect-oriented program when aspects or the base system evolve.

The compile-time feature of the tool is also a drawback. The current implementation is unable to detect and check dynamic join points (for example using the `if` keyword in AspectJ).

4.4.5 Writing specifications in the base system

Developers can specify the base system by using their preferred method. Nevertheless, at the beginning they may find this task difficult. In order to alleviate the charge of writing the specification, I propose the following reference process:

1. Review the places in the base system where ABIS raises warnings.
2. For each warning:
   (a) If the aspect or advice violating the specs is meant to advise the program, then allow it. To do so, add an `@Spect` annotation allowing the violating pattern. If an `@Spect` annotation already exists, then add the pattern to the annotation arguments.
   (b) Otherwise, review the aspect or advice code.
3. If a class contains several occurrences of an (allowed) invasiveness pattern, then allow the pattern at class level. To do so add an `@Spect` annotation allowing the pattern. Forbid the allowed pattern in all the places inside the class where it was forbidden.
4. If required, close the classes, fields or methods to the (by default) allowed invasiveness pattern.
5. Repeat this process each time the base system or the aspects evolve.

How to determine whether an advice must be allowed or forbidden is left to the developer's criterion.

4.5 Experiments

In this section I present the experiment I conduct in order to show the usefulness of the specification framework proposed here. My aim with this experiment is to emphasize the benefits in terms of confidence increment and reduction of the time needed to validate an aspect-oriented program.

The experiment consists in revisiting the case study presented in section 4.2 by specifying the interaction between aspects and base system. Then, I use these specifications to spot and solve the problems that arise after the base system evolution.

4.5.1 Revisiting the initial version

I have specified the initial version of the chat application by following the process presented in section 4.4.5. The annotations I have added allow the invasiveness patterns that interact safely with the base system. Therefore, only the advices realizing the specified patterns can advise the base system.

```
1  public class StandardClientImpl implements IClientIM{
2
3      @Spec(allow={"ArgumentPassing"})
4      public synchronized void transmit(Message message){
5          ...
6      }
7
8      @Spec(allow={"ArgumentPassing"})
9      public void send(String sContents){
10         ...
11     }
12
13     @Spec(allow={"ArgumentPassing"})
14     private void retransmit(Message message){
15         ...
16     }
17     ...
18 }
```

Listing 4.9: `StandarClient` with annotations

Listing 4.8 and 4.9 present the fragment of the specified methods. The methods `send`, `transmit` and `retransmit` were specified to allow the invasiveness pattern *Argument passing*. Aspect and advices were automatically specified by ABIS. Thanks to these specifications, ABIS reports that no aspect is violating the base system specifications. Furthermore, the addition of annotations is transparent for the tests we execute; hence, the test results are not affected.

4.5.2 Evolution, problems detection, and problem solving

After specifying the initial version of the chat application, it is evolved as explained in section 4.2.5. As a result of this evolution, ABIS reports that some aspects are violating the base system specifications. The reports (4.9) indicate that the aspects `EncryptionAspect` and `UniqueName` are violating the specifications on `AuthenticatedClient` (`attach` method) and `AuthenticatedServer` (`send` and `transmit` methods) respectively. This means that potentially unexpected behavior can emerge from the weaving of these invasive aspects advising these new join points.

Using this information we trace unexpected interactions to the violator advices (problems presented in section 4.2.7). Knowing the violating pattern helps us reasoning about the causes of the unexpected interaction, hence, reasoning about the source of the prob-

lem. For example, the violating pattern *Conditional Replacement* tells us that something is wrong because it is possible that a method, which must always execute, sometimes will not be executed.

The further correction of the problems is a developer decision, however, the violated specification may help her finding the solution. In this case, we have applied the same procedure used in section 4.2.7, i.e. change the point-cut descriptors. Once applied the corrections, ABIS reports that no specification is violated and aspects are valid in relation with the base system specifications. Test case where once again executed to verify that the newly modified version was working correct, all test passed.

4.5.3 Benefits for validation and validation effort

Table 4.3: Comparison of the test execution, test time, specifications, and aspect weaving between the specified and non specified chat correction process.

	Not Specified	**Specified**
Tests executed	55	19
Times tested	5	2
Annotations added	0	4
Un-weaved aspects	2	0

Adding interaction specifications reduces the effort and time required to locate faulty aspects because there is no need to successively execute a set of tests. Table 4.3 presents a comparison of different factors involved in the process of correcting the problems due to evolution. The column *Not Specified* shows the values for the correction process through manual inspection and testing of the code. The column *Specified* shows the values when correcting the problem using the specifications as a guide. *Tests executed* corresponds to the total amount of test executed from the initial version to the evolved (corrected) version. *Times tested* corresponds the total amount of times that tests were executed to correcting the problem (all test passes). The difference between the specified and non-specified process is explained by the fact that using specification there is no need to successively execute the tests. Violations on specifications are resolved statically providing information about interaction problems without need to execute. Nevertheless, it is still necessary to execute some tests to validate the initial and the evolved (corrected) version. *Tests executed* and *Times tested* indicate that using specifications reduces the effort involved in testing and tracing an interaction problem. *Annotations added* corresponds to the amount of specifications added to the base system. In the non-specified version there are no specifications, and therefore, no cost associated with specifying the program. The cost involved in adding specifications corresponds to the join points where the invasive aspects or advices advise the base system. We think that the cost of adding specification elements in the program is well rewarded by saving the execution of 36 tests. Finally, *Un-weaved aspects* corresponds

to the amount of aspects that were removed from the application through the correction process. On one hand, the non specified process required to remove 2 aspects to determine whether they were faulty. On the other, no aspect was removed in the specified process because the specifications assisted developers to identify faulty interactions and their possible reasons. The unspecified process adds an additional overhead to the developer that is removing suspect aspects and re-executing the tests. We think that all these reasons justify the effort and time involved in specifying (annotating) the base system.

4.6 Discussion

AOP's problems can hamper software evolution, introduce faulty interactions, and negatively impact the validation of aspect-oriented programs. Through the evolution of an aspect-oriented chat application, I have shown that tracing problem to unexpected interactions is a long and tedious process. Such a process involves rigorous manual inspection of code and the execution of several test scenarios.

In this thesis I tackle AOP's problems by proposing a specification framework for aspect-oriented programs. This framework enables developers to: (1) control the effect of aspects in evolution, (2) control the invasiveness of aspects over the base system, (3) establish an interface that protects critical portions of the base system, (4) reduce the overhead of validating the overall system when new aspects or modifications are introduced, and finally (5) being aware about the existence of aspects and interactions.

Specifying interactions give developers feedback about the harmfulness of aspects. This information assists them in the process of creating an aspect-oriented program, and ensures that aspects perform as expected. The violation of specifications indicates that the violator (aspect or advice) is not meant to advise the code. Therefore, the violator and the affected part of the base system should be reviewed. In this way, developers can be conscious about the aspects they write by enforcing its reasoning of the interactions between aspects and base system.

By specifying and evolving the chat application I have shown that specifications reduce the time and effort needed to locate faults introduced by unexpected interactions. Specifications improve the evolvability, maintainability, and reduce the time and effort needed to validate aspect-oriented program. Since developers can use specifications to ensure that critical parts of the base system will not be modified unexpectedly by future addition of aspects, specifications increase the confidence that developers have on aspects.

For *self*-adaptive systems, these specifications bring multiple benefits. *Self*-adaptation is a general case for evolution, in which the system evolves in order to keep its synchronization with the environment. Every new aspect woven into the system introduces new interactions. Since foreseeing every possible interaction is not always possible (mainly due to the large number of possible weavings, for example the chat application, a small aspect-oriented program, produces $5! = 120$ possible aspect weaving), specifications fit developers

with the capacity to specify locally the intended interactions (with respect to invasiveness) and be notified when something goes wrong. Furthermore, since it is unpractical to execute test in every possible system configuration because of the large number of possible configurations, developers can use specifications to determine whether it is necessary to re-test the system when new aspects are introduced. If invasive aspects (potentially harmful) are introduced, then it may be advisable to test the system and check whether aspects may break it. If developers detect faults, they can use the specifications to locate the faults without re-executing every single test hopping to find the fault's culprit.

Nevertheless, this specification framework and the experiment I conducted are far from being complete. Using this specification framework does not ensure that invasive aspects will perform harmless. It rather ensures that whenever an invasive aspect advises a non-authorized part of the base code, developers will be aware of it. Regarding non-invasive aspects, this specifications framework cannot guarantee that they will not introduce undesired side effect. Yet, since they are non-invasive, I make the hypothesis that if they introduce side effects they will not disturb the underlying core computations performed by the base system.

The experiment I performed was carried out with only one subject, which can be non-representative of existing AspectJ programs. Nonetheless, I believe the techniques used to implement the chat application, and the concerns implemented represent the concerns that are typically encapsulated using aspects. The problems and evolution introduced by the aspects can be perceived as artificial. I think they are not. Developers must write complex PCDs in order to capture the desired join-points. The language proposed by AspectJ is very powerful and complex, and it is easy to write wrong PCDs.

Finally, this specification framework applies to adaptive and non-adaptive software developed with aspects. The concepts and examples I presented in this chapter are in the AspectJ language, however, I believe that they may apply to other AspectJ like languages such as CaesarJ [8].

In this thesis I make the claim that AOP's problems hamper its adoption as an adaptation mechanism. They also hamper AOP's adoption in general. In a recent study [166] I have shown that AOP's invasive features and PCD constructs are scarcely used. Most open-source projects including AOP use similar kind of aspects, which are very precise and not very crosscutting. This gives us a measure of the low adoption of AOP in the open-source developers.

Chapter 5

Conclusion and Perspectives

In this chapter, I draw a series of conclusions from the problems and the solutions introduced in this thesis. I also present the perspectives for future research in validation and verifications of *self*-adaptive systems.

5.1 Summary and Conclusion

Creating, maintaining, and validating *self*-adaptive systems is complex and challenging. In this thesis, I have addressed two of the challenges to validate these complex systems.

In the first part of this thesis I have presented the state of the art of *self*-adaptive systems. I have presented each of the bricks that constitute a self-adaptive system, the techniques and methodologies to construct these bricks, and the current techniques to validate and verify them. I have also pointed out the limitations of existing techniques to validate and verify *self*-adaptive systems in order to motivate my research and to position this thesis relative to the state of the art. In this thesis I focus on the validation of two bricks of *self*-adaptive systems: (i) *reasoning engines*, the decision makers and drivers of the system *self*-adaptation, and (ii) *aspect-oriented programming (AOP) based adaptation mechanism*, a promising technology to realize adaptation mechanisms.

In the second part of this thesis, I have addressed the validation of reasoning engines. First, I have shown that testing whether reasoning engines make the right decision over all the reasoning space is typically unfeasible due to its large size. The reasoning space is characterized by the interactions among reasoning variables. These interactions occur among the values of different reasoning variables (inter-variable interactions, which reflect the different environmental conditions), and between the different values of each reasoning variable (intra-variable interactions, which reflect the environmental variations over time). Next, I have demonstrated that current state of the art technique for sampling large spaces – MCA – can only address one kind of interactions. Nevertheless, to characterize the reasoning space, both inter and intra variable interactions must be covered. Follow-

ing, I have introduced the first contribution of this thesis, a technique – *multi-dimensional covering arrays* (MDCA) – to sample both inter and intra variable interactions at the same time. MDCA defines a set of properties that when satisfied ensure the coverage of inter and intra variable interactions. In order to illustrate the feasibility of MDCA, I have proposed three functional optimization algorithms to construct arrays satisfying the MDCA properties. Finally, I have conducted an empirical study to evaluate the effectiveness of MDCA to cover decisions. In this study I have compared three sampling techniques, MDCA, MCA and random generation. The empirical evidence shows that MDCA effectively covers inter and intra variable interactions, and that a good coverage of such interactions implies good coverage of the reasoning engine's decisions.

In the third part of this thesis, I have addressed AOP's invasiveness, evolution, and interaction problems. AOP has good characteristics to make an adaptation mechanism. It enables developers to augment and modify system's structure and behavior. With AOP, the system's adaptations are encoded into aspects that should modify or reconfigure the system according to the reasoning engine decisions. Nevertheless, AOP has major problems that hamper its adoption as an adaptation mechanism and negatively impacts its validation. Through the evolution of an aspect-oriented chat application I have shown that tracing problems to unexpected aspect interactions is a long and tedious process. Such a process involves rigorous manual inspection of code and the execution of several test scenarios. These evolution problems are due to uncontrolled interactions, invasiveness, obliviousness, and unsupported evolution. In order to address these problems, I introduced the second contribution of this these, a specification framework for interaction between aspects and the base system (ABIS). ABIS automatically specifies the invasiveness patterns that characterize aspects, and enables developers to specify the base system with allowed or forbidden invasiveness patterns. Specifications enable developers to state, in terms of invasiveness patterns, the interactions they want between aspects and the base system. Finally, I have revisited the aspect-oriented chat application by adding interaction specifications and evolving it. This shows how specifications can reduce the cost overhead of validating and finding faults in aspect-oriented programs.

From these contributions I draw the following conclusions:

- **It is feasible to construct data sets that satisfy the MDCA properties. Yet, it is a complex optimization problem.**
- **MDCA effectively covers inter and intra variable interactions, which results in coverage of decisions.**
- **MDCA renders feasible for engineers testing reasoning engines without trying every possible environment condition.**

- **ABIS enables developers to specify what they expect from aspects to do with the base code.**
- **ABIS enables developers to control the invasiveness of aspects and establish an interface that protect critical base system's elements.**
- **ABIS reduces the overhead of validating the system when new aspects or modi-**

fications are introduced.

These contributions sum to the body of knowledge in validating decision-making, self-adaptive systems, and aspect-oriented programs.

5.2 Perspectives

All the pieces of *self*-adaptive systems must be valid from top to bottom. The environment representation and the environmental conditions monitored by sensors must be valid and consistent, otherwise the decisions made by reasoning engines would be based on faulty data. The decisions made by reasoning engines must be correct given any environmental condition, otherwise wrong decisions may leak into the adaptation mechanism and may eventually break the system down. Finally, adaptation mechanisms and the underlying executing platform must be valid, functional, and robust. The adaptation mechanism must be capable of applying the reasoning engine's decisions correctly. The reconfigurations introduced by the adaptation mechanism into the running platform must not break down the system or prevent it from providing its functionalities.

In this thesis, I have contributed to the validation of two dimensions of *self*-adaptive systems. Nevertheless, these contributions are only a small part of the big picture and leave room for improvement and future research. In this section, I present some of the perspectives in future research for the validation of *self*-adaptive systems. More precisely, I discuss the perspective for the validation of reasoning engines and adaptation mechanisms.

5.2.1 Perspectives for Reasoning Engine Validation

Reasoning engines occupy an important place in *self*-adaptive system. I present three perspectives for future work on the validation of reasoning engines.

Definition of automated oracles: How to know when the decisions made by the reasoning engine are right without requiring engineers to check the decisions? This question poses a big challenge to researchers, engineers, and testers. Currently, I make the hypothesis that engineers judge the reasoning engine's decisions. This is a tedious, long, and complex process. The automation of the oracle should enable a faster and more reliable validation of reasoning engines. Initial ideas on how to define these oracles consist in evaluating whether the system continues working after an environmental change, or whether it continues to provide a certain number of functionalities. Assertion languages may allow defining patterns that decisions must contain in order to be valid.

Reasoning engine validation at runtime: Due to the huge size of the reasoning space, it is impossible to validate reasoning engines against all possible environmental conditions. An option to explore the environmental conditions not covered by MDCA (and that could still trigger faults) consists in testing the system as it runs. Validation at runtime refers to online or in vivo testing [46], whose underlying idea is to select environmental conditions

that are likely to occur soon. These conditions are then fed to a sand-boxed version of the reasoning engine, which makes a decision. Then, an automated oracle evaluates if the decision is right. If a wrong decision is made, the executing reasoning engine is patched with fault corrections on the fly. Notice that such testing techniques require the development of oracles with the capability of automatically evaluating the correctness of the reasoning engine's decision at runtime. Validation at runtime may also refer to a combination of online verification and testing. Several sand-boxed copies of a reasoning engine may explore through online test different portions of the reasoning space, whereas a verification procedure may check whether online corrections do not break down the reasoning engine. Furthermore, the validation at runtime could benefit from the emerging cloud services. Testing and verification could be performed remotely by a farm of services in the cloud, error detection and corresponding patches distributed to different instances of reasoning engines.

Validation for stochastic non-deterministic reasoning engines: In recent years, several researchers have proposed reasoning engines and reasoning mechanisms based on stochastic models [35, 75]. The main difference between these reasoning engines and traditional rule-based and goal-based engines is that they are non-deterministic. Fairly small variation on their parameters may induce completely different decisions. Validating whether these reasoning engines make the right decision is challenging because there is not *one*, but *many* possible right decisions for a given environmental condition.

5.2.2 Perspectives for Adaptation Mechanism Validation

Model validation for model driven adaptation: In the last years, researchers have explored the usage of models to leverage the potential of a higher abstraction level. Model composition plays an important role in model driven adaptation, and since it is analogous to AOP based adaptation mechanism, it allows defining the adaptive concerns as architectural changes encapsulated inside aspect models. These models are then woven into the base model to produce the adapted architecture. Yet, aspects and base models must be faultless in order to allow the system to adapt. If aspects or base models contain faults, their composition will result in unexpected and undesired consequences that may prevent the system to adapt (or even break it down). Furthermore, the model composition engines used to compose aspects and base models can also introduce faults and produce faulty models. Model composition engines must be tested in order to ensure that the models resulting from the composition are as expected. Nevertheless, testing such programs is challenging because of the complexity of the models they manipulate. I have addressed the first steps in testing model composition engines [163], however, there are still many issues to address.

System validation at runtime: Over recent years, the use of models at runtime has gained a large number of followers [20]. The use of models at runtime represents an important part of the effort to drive the adaptation process using models [158, 155, 157, 156].

These models at runtime can be leveraged to perform validation and verification activities at runtime. Similar to the sand-boxing of reasoning engines, models at runtime provide a platform to simulate, verify, and test configurations and adaptations before applying them to the running platform. Models describing the functioning of the running system could be used to verify whether the selected adaptation procedure is optimal, or whether the adaptation will preserve the system functionalities. Models and simulation could even be used to explore future reconfigurations and give feedback information to the reasoning engine.

Appendix A

Tuple counting procedure

The procedure on Listing A.1 counts the number of different t-tuples of u-consecutive columns contained by an array X with k rows. Basically, the procedure assembles the t-tuples of u-consecutive columns and stores them inside a data structure. Notice that the procedure stores only the t-tuples that have not being stored before.

```
1  procedure count_tuple( X, u, t, k): returns integer
2      L: list of list containing tuples
3      T: list containing tuples
4      count, i, h, s, j, n: integer
5      n = length of N
6      Y: array
7      for i = 1 until i = n − u
8          // Y: array of size k x u
9          Y = X [ ] [ i to i + u]
10         for h = 1 until h = k − t
11             // Y [ h ] array of length 1 x u
12             add Y [ h ] to T
13             for s = h + 1 until s = k
14                 for j = 1 until j = t
15                     add Y [ s + j ] to T
16                 end
17                 // if the list L contains the tuple T
18                 if T not contained in L then
19                     count = count + 1
20                     // add the tuple T to the list L
21                     add T to L
22                 end
```

```
23                  // remove all the elements of T
24                  clear T
25              end
26          end
27      end
28      return count
29  end
```

Listing A.1: Tuple counting procedure

Appendix B

Authentication mechanism for the chat application using aspects

The evolution or adaptation of the aspect-oriented chat application described in Section 4.2.5 can be implemented either using plain Java (static at compile time), or using AspectJ aspects (which can eventually be dynamically woven / unwoven). This appendix presents the AspectJ code that realizes using AOP the evolution of the aspect-oriented chat application introduced in Section 4.2.1.

The new concern to introduce is *authentication*, which allows the chat application to accept connections from users identified with a *username* and a *password*. Listing B.1 and B.2 present the AspectJ code for the authentication adaptation for client and server. The code in Listing B.1 illustrates the `AuthenticatedServerAspect`, which introduced the authentication concern by adding new operations to the `StandardServerImpl` class (cf. Section 4.2.1). Additionally, an around advice intercept the `attach` method and verifies whether the client *username* and *password* are accepted in the server. Analogously, the code in Listing B.1 adds a new `initialize` method that accepts a new argument (*password*). This method stores the client password and then continues the original computation.

```
1  package chat.aspect;
2  import java.io.BufferedReader;
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.io.FileReader;
6  import java.io.IOException;
7  import java.rmi.RemoteException;
8  import java.util.Hashtable;
9  import chat.impl.StandardServerImpl;
10 import chat.support.Client;
```

```
11
12  public aspect AuthenticatedServerAspect {
13
14      private Hashtable<String,String> StandardServerImpl.registeredUser
15                                          = new Hashtable<String, String>();
16
17      private void StandardServerImpl.loadUsers(){
18          try {
19              File file=new File("pwd.txt");
20              BufferedReader input
21                              = new BufferedReader(new FileReader(file));
22
23                  String line = null;
24                  while (( line = input.readLine()) != null){
25                      String[] ps=line.split("::");
26                      registeredUser.put(ps[0], ps[1]);
27                  }
28                  input.close();
29          } catch (FileNotFoundException e) {
30              e.printStackTrace();
31          } catch (IOException e) {
32              e.printStackTrace();
33          }
34      }
35
36      private boolean StandardServerImpl.authenticate(Client client){
37          if(this.registeredUser.get(client.getSName()).equals(client.getPwd())){
38              return true;
39          }
40          return false;
41      }
42
43      after(StandardServerImpl impl):
44          execution(* StandardServerImpl(..)) && target(impl){
45          impl.loadUsers();
46      }
47
48      int around(Client client,StandardServerImpl impl)
49              throws RemoteException :
50                  execution(int StandardServerImpl.attach(Client))
51                      && args(client) && this(impl){
```

```
52            if(!impl.authenticate(client)){
53                return -1;
54            }
55            return impl.attach(client);
56        }
57    }
```

Listing B.1: Server authentication mechanism implemented using aspects

```
1   package chat.aspect;
2   import java.text.SimpleDateFormat;
3   import chat.impl.StandardClientImpl;
4   import chat.support.Message;
5
6   public privileged aspect AuthenticatedClientAspect {
7
8       public boolean StandardClientImpl.initialize(String server,
9                                                    String name, String pwd) {
10          this.m_client.setPwd(pwd);
11          return this.initialize(server, name);
12      }
13
14      before(Message message):
15          execution(void StandardClientImpl.transmit(Message))
16          && args(message){
17          message.
18              setMeta(new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").
19                                      format(new java.util.Date()));
20      }
21  }
```

Listing B.2: Client authentication mechanism implemented using aspects

Appendix C

Alloy Specifications

Listing C.1, presents the formalization in Alloy [115] of the ABIS specification propagation and specification matching strategy. Notice that this specification is meant to check whether the specification propagation strategies are consistent and well formed.

```
1  module abis
2
3  /*
4   * Model of ABIS.
5   */
6
7  abstract sig NameElement {
8       name : one Name
9  }
10 sig Name{}
11
12 abstract sig SpecElement extends NameElement {
13      forbiddenAdviceType,allowedAdviceType : set AdviceType,
14      allForbiddenAdviceType,allAllowedAdviceType : set AdviceType,
15 }
16
17 sig Class extends SpecElement{
18      superclass : set Class,
19      properties : set Property,
20      operations : set Operation
21 }
22
23 sig Property,Operation extends SpecElement{}
```

```
24
25 abstract sig AdviceType {}
26
27 one sig Augmentation, Replacement,
28         ConditionalReplacement, Multiple,
29         Crossing,Write, Read,ArgumentPassing,
30         Hierarchy, FieldAddition,OperationAddition
31         extends AdviceType {}
32
33 fun getAllAdviceType() : set AdviceType{
34     Augmentation + Replacement + ConditionalReplacement
35     + Multiple + Crossing + Write + Read +
36     ArgumentPassing + Hierarchy + FieldAddition + OperationAddition
37 }
38
39 fact ForbiddenTypeClassDerived{
40     all c: Class | all x : c.^superclass | x in Temp.allSuperClasses
41
42     /* No spec */
43     and ((#(Temp.allSuperClasses.forbiddenAdviceType) = 0 )
44     and (#( c.forbiddenAdviceType) = 0 )
45     and (#(Temp.allSuperClasses.allowedAdviceType) = 0 )
46     and (#( c.allowedAdviceType) = 0 ))
47         implies c.allForbiddenAdviceType = getAllAdviceType
48
49     /* Only allowed spec */
50     and (((#(Temp.allSuperClasses.forbiddenAdviceType) = 0 )
51     and (#( c.forbiddenAdviceType) = 0 ))
52     and ((#(Temp.allSuperClasses.allowedAdviceType) != 0 )
53     or (#( c.allowedAdviceType) != 0 )))
54         implies c.allForbiddenAdviceType =
55             ( getAllAdviceType
56                 - (Temp.allSuperClasses.allowedAdviceType + c.allowedAdviceType)
57             )
58
59     /* Only forbidden spec */
60     and (((#(Temp.allSuperClasses.forbiddenAdviceType) != 0 )
61     or (#( c.forbiddenAdviceType) != 0 ))
62     and ((#(Temp.allSuperClasses.allowedAdviceType) = 0 )
63     and (#( c.allowedAdviceType) = 0 )))
64         implies c.allForbiddenAdviceType =
```

```
65                Temp.allSuperClasses.forbiddenAdviceType + c.forbiddenAdviceType
66
67        /* Mix */
68        and (((#(Temp.allSuperClasses.forbiddenAdviceType) != 0 )
69        or (#( c.forbiddenAdviceType) != 0 ) )
70        and ((#(Temp.allSuperClasses.allowedAdviceType) != 0 )
71        or (#( c.allowedAdviceType) != 0 )))
72            implies c.allForbiddenAdviceType =
73                ( getAllAdviceType -
74                    (Temp.allSuperClasses.allowedAdviceType + c.allowedAdviceType)\
75                )
76                + Temp.allSuperClasses.forbiddenAdviceType + c.forbiddenAdviceType
77        all c: Class | c.allAllowedAdviceType = getAllAdviceType - c.allForbiddenAdviceType
78    }
79
80
81
82    fact ForbiddenTypePropertyDerived{
83        all p: Property | all x : p.~properties.^superclass| x in Temp.allSuperClassesProperty
84        and all cps : Temp.allSuperClassesProperty.properties | cps in Temp.allSuperProperties
85
86        /* No spec */
87        and ((#(Temp.allSuperProperties.forbiddenAdviceType) = 0 )
88        and (#( p.~properties.allForbiddenAdviceType) = 0 )
89        and (#( p.forbiddenAdviceType) = 0 )
90        and (#(Temp.allSuperProperties.allowedAdviceType) = 0 )
91        and (#( p.~properties.allAllowedAdviceType) = 0 )
92        and (#( p.allowedAdviceType) = 0 ))
93            implies p.allForbiddenAdviceType = getAllAdviceType
94
95        /* Only allowed spec */
96        and (((#(Temp.allSuperProperties.forbiddenAdviceType) = 0 )
97        and (#(Temp.allSuperClassesProperty.allForbiddenAdviceType) = 0 )
98        and (#( p.forbiddenAdviceType) = 0 ))
99        and ((#(Temp.allSuperProperties.allowedAdviceType) != 0 )
100       or (#(Temp.allSuperClassesProperty.allAllowedAdviceType) != 0 )
101       or (#( p.allowedAdviceType) != 0 )))
102           implies p.allForbiddenAdviceType =
103           (getAllAdviceType -
104               (Temp.allSuperProperties.allowedAdviceType
105                   + p.allowedAdviceType+ p.~properties.allAllowedAdviceType
```

```
106                 )
107             )
108
109         /* Only forbidden spec */
110         and (((#(Temp.allSuperProperties.forbiddenAdviceType) != 0 )
111         or (#( p.~properties.allForbiddenAdviceType) != 0 )
112         or (#( p.forbiddenAdviceType) != 0 ))
113         and ((#(Temp.allSuperProperties.allowedAdviceType) = 0 )
114         and (#( p.~properties.allAllowedAdviceType) = 0 )
115         and (#( p.allowedAdviceType) = 0 )))
116             implies p.allForbiddenAdviceType =
117                 Temp.allSuperProperties.forbiddenAdviceType
118                 + p.forbiddenAdviceType+ p.~properties.allForbiddenAdviceType
119
120         /* Mix */
121         and (((#(Temp.allSuperProperties.forbiddenAdviceType) != 0 )
122         or (#( p.~properties.allForbiddenAdviceType) != 0 )
123         or (#( p.forbiddenAdviceType) != 0 ))
124         and ((#(Temp.allSuperProperties.allowedAdviceType) != 0 )
125         or (#( p.~properties.allAllowedAdviceType) != 0 )
126         or (#( p.allowedAdviceType) != 0 )))
127             implies p.allForbiddenAdviceType =
128                 (getAllAdviceType -
129                     (Temp.allSuperProperties.allowedAdviceType
130                         + p.allowedAdviceType + p.~properties.allAllowedAdviceType)
131                 )
132                 + Temp.allSuperProperties.forbiddenAdviceType
133                 + p.forbiddenAdviceType
134                 + p.~properties.allForbiddenAdviceType
135         all p: Property | p.allAllowedAdviceType = getAllAdviceType - p.allForbiddenAdviceType
136 }
137
138
139
140 fact ForbiddenTypeOperationDerived{
141         all o: Operation | all x : o.~operations.^superclass | x in Temp.allSuperClassesOperation and
142         all cps : Temp.allSuperClassesOperation.operations | cps in Temp.allSuperOperations
143
144         /* No spec */
145         and ((#(Temp.allSuperProperties.forbiddenAdviceType) = 0 )
146         and (#(o.~operations.allForbiddenAdviceType) = 0 )
```

```
147     and (#( o.forbiddenAdviceType) = 0 )
148     and (#(Temp.allSuperProperties.allowedAdviceType) = 0 )
149     and (#( o.~operations.allAllowedAdviceType) = 0 )
150     and (#( o.allowedAdviceType) = 0 ))
151         implies o.allForbiddenAdviceType = getAllAdviceType
152
153     /* Only allowed spec */
154     and (((#(Temp.allSuperProperties.forbiddenAdviceType) = 0 )
155     and (#( o.~operations.allForbiddenAdviceType) = 0 )
156     and (#( o.forbiddenAdviceType) = 0 ))
157     and ((#(Temp.allSuperProperties.allowedAdviceType) != 0 )
158     or (#( o.~operations.allAllowedAdviceType) != 0 )
159     or (#( o.allowedAdviceType) != 0 )))
160         implies o.allForbiddenAdviceType =
161         (getAllAdviceType -
162             (Temp.allSuperProperties.allowedAdviceType
163                 + o.allowedAdviceType+o.~operations.allAllowedAdviceType)
164         )
165     /* Only forbidden spec */
166     and (((#(Temp.allSuperProperties.forbiddenAdviceType) !=0 )
167     or (#( o.~operations.allForbiddenAdviceType) !=0 )
168     or (#( o.forbiddenAdviceType) !=0 ))
169     and ((#(Temp.allSuperProperties.allowedAdviceType) =0 )
170     and (#( o.~operations.allAllowedAdviceType) =0 )
171     and (#( o.allowedAdviceType) =0 )))
172         implies o.allForbiddenAdviceType =
173             Temp.allSuperProperties.forbiddenAdviceType
174             + o.forbiddenAdviceType
175             + o.~operations.allForbiddenAdviceType
176     /* Mix */
177     and (((#(Temp.allSuperProperties.forbiddenAdviceType) != 0 )
178     or (#( o.~operations.allForbiddenAdviceType) !=0 )
179     or (#( o.forbiddenAdviceType) !=0 ))
180     and ((#(Temp.allSuperProperties.allowedAdviceType) !=0 )
181     or (#( o.~operations.allAllowedAdviceType) !=0 )
182     or (#( o.allowedAdviceType) !=0 )))
183         implies o.allForbiddenAdviceType =
184             (getAllAdviceType -
185                 (Temp.allSuperProperties.allowedAdviceType
186                     + o.allowedAdviceType
187                     + o.~operations.allAllowedAdviceType)
```

```
188                 )
189                 + Temp.allSuperProperties.forbiddenAdviceType
190                 + o.forbiddenAdviceType
191                 + o.~operations.allForbiddenAdviceType
192     all o: Operation | o.allAllowedAdviceType = getAllAdviceType - o.allForbiddenAdviceType
193 }
194
195
196 sig Aspect extends NameElement{
197     advices : set Advice
198 }
199
200 sig Advice extends NameElement{
201     joinpointop : set SpecElement,
202    type : one AdviceType
203 }
204
205 fact adviceRight{
206     all a : Advice | all c : SpecElement | a.type not in c.allForbiddenAdviceType
207         and a.type in c.allAllowedAdviceType
208 }
209
210 assert ConjunctionClasses{
211     all C : Class | C.allForbiddenAdviceType + C.allAllowedAdviceType = getAllAdviceType
212 }
213 assert DisjunctionClasses{
214     all C : Class | #(C.allForbiddenAdviceType - C.allAllowedAdviceType) = 0
215 }
216
217 assert ConjunctionOperations{
218     all o : Operation | o.allForbiddenAdviceType
219                 + o.allAllowedAdviceType = getAllAdviceType
220 }
221
222 assert DisjunctionOperations{
223     all o : Operation | #(o.allForbiddenAdviceType - o.allAllowedAdviceType) = 0
224 }
225
226 assert ConjunctionProperties{
227     all p : Property | p.allForbiddenAdviceType
228                 + p.allAllowedAdviceType = getAllAdviceType
```

```
229 }
230
231 assert DisjunctionProperties{
232     all p: Property | #(p.allForbiddenAdviceType − p.allAllowedAdviceType) = 0
233 }
234
235 sig Temp
236 {
237     allSuperClasses : set Class,
238     allSuperClassesProperty : set Class,
239     allSuperProperties : set Property,
240     allSuperClassesOperation : set Class,
241     allSuperOperations : set Operation
242 }
243
244 assert AspectCanNotBe{
245     all a : Aspect | all c : SpecElement | all adv : Advice | adv in a.advices and
246                 #(c.allAllowedAdviceType ) − #(c.allAllowedAdviceType− adv.type)=1
247 }
248
249 check ConjunctionClasses for 30
250 check DisjunctionClasses for 30
251
252 check ConjunctionOperations for 20
253 check ConjunctionProperties for 20
254 check DisjunctionOperations for 20
255 check DisjunctionProperties for 20
256
257 check AspectCanNotBe for 20
```

Listing C.1: Alloy specification of the ABIS framework

Glossary ‡

Customizability The ability for software to be changed by the user or programmer.

Environment In *self*-adaptive system, the *environment* refers to the surroundings of the system. These can be composed of a mixture of physical and virtual properties. Sometimes the environment is also referred as working environment or working context.

Flexibility The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

Human-like reasoning The capacity to spark creative and working solution to complex problems. In self-adaptive systems is the capacity of reasoning engines to

LOC Acronym for lines of code.

Oracle A mechanism used by software testers and software engineers for determining whether a test has passed or failed. It is used by comparing the output(s) of the system under test, for a given test case input, to the outputs that the oracle determines that product should have. Oracles are always separate from the system under test.

Problem slang Is the vocabulary use to communicate terms, actions, elements, etc of a particular problem's domain.

Resiliency The ability for software to recover from a failure or malfunction and continue to work. Sometimes it is referred as fault tolerance or self-healing.

Robustness Is the ability of a computer system to cope with errors during execution or the ability of an algorithm to continue to operate despite abnormalities in input, calculations, etc.

Recoverability Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it.

Validation Is the certification that an information system has been implemented correctly and that it conforms to the functional specifications derived from the original requirements.

Verification Is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

Versatility The ability for software to be applicable and customizable to do many things competently.

Bibliography

[1] Event Stream Intelligence with Esper and NEsper. Codehaus. Website: http://esper.codehaus.org/.

[2] *Systems and software verification: model-checking techniques and tools*. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

[3] *The CRM handbook: a business guide to customer relationship management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] A.D.Precce, C.Grossner, P.G.Chander, and T.Radhakrishnan. Structural validation of expert systems using a formal model. In *Working Notes: Workshop on VV of KBS at AAAI 93, Washington D.C.*, 1993.

[5] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *ECOOP'05: Proceedings 19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, July 2005.

[6] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, page 21. 1998.

[7] Jesper Andersson, Rogerio de Lemos, Sam Malek, and Danny Weyns. Reflecting on self-adaptive software systems. In *SEAMS '09: Proceedings of the International workshop on Software Engineering for Adaptive and Self-Managing Systems in conjunction with ICSE'09*, pages 38–47, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[8] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of caesarj. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.

[9] Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. MOCAS: A State-Based Component Model for Self-Adaptation. *SASO'09: 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 206–215, 2009.

[10] Valerie Barr. Applications of rule-base coverage measures to expert system evaluation. In *Journal of Knowledge-Based Systems*, volume 12, pages 27–35. Elsevier, 1998.

[11] Thaìs V. Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA'05: 2nd European Workshop on Software Architecture*, pages 1–17. Springer, 2005.

[12] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le-Traon. Automatic test cases optimization: a bacteriologic algorithm. *IEEE Software*, 22(2):76–82, 2005.

[13] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 72–81, New York, NY, USA, 2006. ACM.

[14] Boris Beizer. *Software testing techniques*. International Thomson Computer Press, New York, NY, USA, 2nd edition edition, 1990.

[15] N. Bencomo, G. Blair, G. Coulson, and T.V. Batista. Towards a Meta-Modelling Approach to Configurable Middleware. In *RAM-SE'05: Workshop on Reflection, AOP, and Meta-Data for Software Evolution in conjuction with ECOOP'05*, Lecture Notes in Computer Science, pages 73–82. Springer, July 2005.

[16] Nelly Bencomo. *Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability*. PhD thesis, Lancaster University, March 2008.

[17] Nelly Bencomo. On the use of software models during software execution. In *MISE'09: Proceedings of the Workshop on Modeling in Software Engineering in conjuction with ICSE'09*, New York, NY, USA, May 2009. ACM.

[18] Nelly Bencomo and Gordon Blair. Using architecture models to support the generation and operation of component-based adaptive systems. *Software Engineering for Self-Adaptive Systems*, 5525:183–200, June 2009.

[19] Nelly Bencomo, Gordon Blair, Geoff Coulson, Paul Grace, and Awais Rashid. Reflection and aspects meet again: Runtime reflective mechanisms for dynamic aspects. In *First Middleware Workshop on Aspect Oriented Middleware*, New York, NY, USA, 2005. ACM.

[20] Nelly Bencomo, Gordon Blair, Robert France, Freddy Munoz, and Cedric Jeanneret. Proceedings of the models@run.time workshops 2007, 2008, 2009. In *International conference on Model Driven Engineering Languages and Systems*. Springer, 2007 - 2009. www.comp.lancs.ac.uk/ bencomo/MRT/.

[21] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *ICSE '08: Proceedings of the 30th international conference on Software engineering, Formal Research Demonstrations Track*, pages 811–814, New York, NY, USA, May 2008. ACM.

[22] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1999.

[24] J. Bloch. A metadata facility for the java programming language. Technical report jsr 175, Sun Microsystems, www.jcp.org, 2002.

[25] Jonas Bonér and Alexandre Vasseur. Aspectwerkz. http://aspectwerkz.codehaus.org/index.html, February 2004.

[26] Rafael H. Bordini, Louise A.Dennis, Berndt Farwer, and Michael Fisher. Automated verification of multi-agent programs. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 69–78, Washington, DC, USA, 2008. IEEE Computer Society.

[27] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic management of clustered applications. In *Cluster'06: Proceedings of the 2006 IEEE International Conference on Cluster Computing*, pages 25–28, Washington, DC, USA, September 2006. IEEE Computer Society.

[28] Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Adrian Mos, Noel de Palma, Vivien Quema, and Jean-Bernard Stefani. Architecture-based autonomous repair management: Application to j2ee clusters. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 369–370, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Ivan Bratko. *PROLOG Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[30] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. pages 48–70, 2009.

[31] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software Practice Experts*, 36(11-12):1257–1284, 2006.

[32] Renée C. Bryce and Charles J. Colbourn. Constructing interaction test suites with greedy algorithms. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 440–443, New York, NY, USA, 2005. ACM. 1101994 440-443.

[33] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly, 2006.

[34] Kevin Burr and William Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513. West, 1998.

[35] Radu Calinescu and Marta Kwiatkowska. Using quantitative analysis to implement autonomic it systems. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 100–110, Washington, DC, USA, 2009. IEEE Computer Society.

[36] Cetina Carlos, Haugen Øystein, Zhang Xiaorui, and Fleurey Franck. Strategies for variability transformation at runtime. In *SPLC'09: 13th International Software Product Line Conference*, pages 61–70, New York, NY, USA, 2009. ACM.

[37] Carlos Cetina, Joan Fons, and Vicente Pelechano. Applying Software Product Lines to Build Autonomic Pervasive Systems. In *SPLC'08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 117–126, Limerick, Ireland, 2008. IEEE Computer Society.

[38] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *IEEE Computer*, 42:37–43, 2009.

[39] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Using feature models for developing self-configuring smart homes. In *ICAS'09: Proceedings of the 2009 Fifth International Conference on Autonomic and Autonomous Systems*, pages 179–188, Washington, DC, USA, April 2009. IEEE Computer Society.

[40] F. Chauvel, O. Barais, J.M. Jézéquel, and I. Borne. A model-driven process for self-adaptive software. In *ERTS'08: 4th European Congress on Embedded Real Time Software*, Toulouse, France, 2008.

[41] Frank Chauvel, Olivier Barais, Isabelle Borne, and Jean-Marc Jézéquel. Composition of qualitative adaptation policies. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 455–458, Washington, DC, USA, 2008. IEEE Computer Society.

[42] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical report, Hanover, NH, USA, 2000.

[43] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, Berlin, Heidelberg, 2009.

[44] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. *Software Engineering for Self-Adaptive Systems*, 5525:1–26, 2009.

[45] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems in conjunction with ICSE'06*, pages 2–8, New York, NY, USA, 2006. ACM.

[46] Matt Chu, Christian Murphy, and Gail Kaiser. Distributed in vivo testing of software applications. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:509–512, 2008.

[47] Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley Professional, April 2005.

[48] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition edition, August 2001. 501065.

[49] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Ron Cytron and Gary T. Leavens, editors, *FOAL'02: Foundations of Aspect-Oriented Languages in conjunction with AOSD-2002*, pages 33–44, March 2002.

[50] David M. Cohen, R. Dalal Siddhartha, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, vol.23(7):437–444, 1997. 264150.

[51] Myra B. Cohen, Charles J. Colbourn, and Alan C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 394–405, Washington, DC, USA, 2003. IEEE Computer Society.

[52] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63, New York, NY, USA, 2006. ACM.

[53] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM. 1273482.

[54] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[55] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Myra B. Cohen, Joshua Snyder, and Gregg Rothermel. Testing across configurations: implications for combinatorial testing. *ACM SIGSOFT Software Engineering Notes*, 31(0163-5948):1–9, 2006.

[57] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The runes middleware: A reconfigurable the runes middleware for networked embedded systems and its application in a disaster management scenario. In *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, pages 69–78, Washington, DC, USA, 2007. IEEE Computer Society.

[58] Stefania Costantini. Meta-reasoning: A survey. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 253–288, London, UK, 2002. Springer-Verlag.

[59] Geoff Coulson, Gordon S. Blair, Mike Clark, and N. Parlavantzas. The design of a highly configurable and reconfigurable middleware platform. *ACM Distributed Computing Journal*, 15(2):109–126, 2002.

[60] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[61] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[62] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[63] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC'05: 3rd Int. Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 1–7, New York, NY, USA, 2005. ACM.

[64] Pierre-Charles David and Thomas Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *SC'06: 5th International Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97, London, UK, 2006. Springer-Verlag.

[65] Pierre-Charles David and Thomas Ledoux. Safe Dynamic Reconfigurations of Fractal Architectures with FScript. In *Proceeding of Fractal CBSE Workshop in conjuction with ECOOP'06*, London, UK, 2006. Springer-Verlag.

[66] Pierre-Charles David, Marc Léger, Hervé Grall, Thomas Ledoux, and Thierry Coupaye. A multi-stage approach for reliable dynamic reconfigurations of component-based systems. In *DAIS'08: 8th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, pages 106–111, London, UK, 2008. Springer-Verlag.

[67] Randall Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15(3):179 – 222, 1980.

[68] Hamlet Dick. Software component composition: a subdomain-based testing-theory foundation. *Software Testing Verification and Reliability*, 17(0960-0833):243–269, 2007.

[69] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.

[70] Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-model for Self-Adaptive Software. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[71] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, June 2008.

[72] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in jpl's mission data system. In *IEEE Aerospace Conference Proceedings*, volume 7, pages 259–268, Washington, DC, USA, 2000. IEEE Press.

[73] Adaptive Systems (DiVA) Work Package 6 Dynamic Variability in Ccomplex. Deliverable 6.1: Case study specification and requirements, 2009.

[74] Frank Eliassen, Gjrven Eli, Viktor S. Wold Eide, and Michaelsen Jorgen Andreas. Evolving self-adaptive services using planning-based reflective middleware. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware*, page 1, New York, NY, USA, 2006. ACM. 1175856 1.

[75] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE'09: 31th International Conference on Software Engineering*, Los Alamitos, CA, USA, pages 111–121, Washington, DC, USA, May 2009. IEEE Computer Society.

[76] Paolo Falcarin and Gustavo Alonso. Software architecture evolution through dynamic aop. In *EWSA' 04: First European Workshop on Software Architecture*, pages 57–73, London, UK, 2004. Springer-Verlag.

[77] Paolo Falcarin and Marco Torchiano. Automated reasoning on aspects interactions. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 313–316, Washington, DC, USA, 2006. IEEE Computer Society.

[78] William Feller. *An Introduction to Probability Theory and Its Applications, Vol. 1, 3rd Edition*. Wiley, 3 edition, January 1968.

[79] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodologies*, 5(1):63–86, 1996.

[80] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[81] Franck Fleurey and Arnor Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *MoDELS'09: Model Driven Engineering Languages and Systems, 12th International Conference. Proceedings*, pages 606–621, London, UK, 2009. Springer-Verlag.

[82] Jacqueline Floch. Theory of adaptation – deliverable d2.2 MUSIC EU project. http://www.ist-music.eu/MUSIC/docs/MADAMdf/view, 2006.

[83] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[84] Kenneth D. Forbus and Johan de Kleer. *Building problem solvers*. MIT Press, Cambridge, MA, USA, 1993.

[85] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[86] Sandro Fouché, Myra B. Cohen, and Adam Porter. Incremental covering array failure characterization in large configuration spaces. In *ISSTA '09: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 177–188, New York, NY, USA, July 2009. ACM.

[87] Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 253, Washington, DC, USA, 2007. IEEE Computer Society.

[88] Christian Fritz and Sheila A. McIlraith. Decision-theoretic golog with qualitative preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, pages 153–163, Lake District, UK, June 2006.

[89] Bridging The Gap, W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection : Bridging the gap between a running system and its architectural specification. In *Proceedings of the 2 nd Euromicro Conference on Software Maintenance and Reengineering and 6 th Reengineering Forum*, pages 8–11, Washington, DC, USA, March 1998. IEEE Computer Society.

[90] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[91] John C. Georgas and Richard N. Taylor. Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems in conjunction with ICSE'08*, pages 105–112, New York, NY, USA, 2008. ACM.

[92] John C. Georgas, Andre van der Hoek, and Richard N. Taylor. Using architectural models to manage and visualize runtime adaptation. *IEEE Computer*, 42(9):52–60, 2009.

[93] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[94] Jacques Klein Benoit Baudry Gilles Perrouin, Sagar Sen and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST'10: Third International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, 2010. IEEE Computer Society.

[95] Allen Ginsberg, editor. *Automatic refinement of expert system knowledge bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[96] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[97] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, and Peter Toft. Smartfrog: Configuration and automatic ignition of distributed applications. Technical report, HP Labs, Bristol, UK, 2003.

[98] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, February 2006.

[99] Clifford Grossner, Alun D. Preece, P. Gokul Chander, Thiruvengadam Radhakrishnan, and Ching Y. Suen. Exploring the structure of rule based systems. In *AAAI'93: Proc. 11th. National Conference on Artificial Intelligence*, pages 704–709, Washington, DC, 1993. AAAI.

[100] Uma G. Gupta. Automatic tools for testing expert systems. *Communications of the ACM*, pages 179–184, 1998.

[101] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4), April 2008.

[102] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *SPLC'06: 10th Internationa Software Product Line Conference*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[103] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information Software Technology*, 43(14):833–839, 2001.

[104] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–156, 2004.

[105] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A case study in goal-driven architectural adaptation. *Software Engineering for Self-Adaptive Systems*, 5525:109–127, 2009.

[106] Lu Heng, W. K. Chan, and T. H Tse. Testing context-aware middleware-centric programs: a data flow approach and an rfid-based experimentation. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 61–70, New York, NY, USA, 2006. ACM.

[107] Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In *ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet*, pages 122–133, Berlin, Heidelberg, 2008. Springer-Verlag.

[108] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.

[109] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM. "976276 26-35".

[110] Koen Hindriks, Mark d'Inverno, and Michael Luck. Architecture for agent programming languages. In *In ECAI'00: Proceedings of the Fourteenth European Conference on Artificial Intelligence*, pages 363–367. Springer-Verlag, 2000.

[111] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3):1–28, 2008.

[112] Danny Hughes, Phil Greenwood, Geoff Coulson, Gordon Blair, Florian Pappenberger, Paul Smith, and Keith Beven. Gridstix: Supporting flood prediction using embedded hardware and next generation grid middleware. In *MDC'06: 4th International Workshop on Mobile Distributed Computing in conjunction with the 7th IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks*, Washington, DC, USA, 2006. IEEE Computer Society.

[113] ILog and IBM Corporation. Jrules. http://www-01.ibm.com/software/integration/business-rule-management/jrules/, 2009.

[114] An intelligent, adaptable grid-based flood monitoring, and warning system. Danny hughes and phil greenwood and gordon blair and geoff coulson and paul smith and keith beven. In *Proceedings of the 5th UK eScience All Hands Meeting*, 2006.

[115] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[116] JBoss and RedHat Corporation. Drools. http://www.jboss.org/drools/, 2009.

[117] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, may 2008.

[118] Bryan F. Jones, Harmen-H. Sthamer, and David E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, September 1996.

[119] Yan Jun and Zhang Jian. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Washington, DC, USA, 2006. IEEE Computer Society.

[120] Edward Kachinske and Timothy Kachinske. *Maximizing Your Sales with Microsoft Dynamics CRM*. Course Technology Press, Boston, MA, United States, 2008.

[121] Emilia Katz and KKatz. Incremental analysis of interference among aspects. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 29–38, New York, NY, USA, 2008. ACM.

[122] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages workshop in conjunction with AOSD'04*, pages 1–6, March 2004.

[123] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–14, Washington, DC, USA, 2003. IEEE Computer Society. 826854 3.

[124] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[125] Jeffrey O. Kephart and Rajarshi Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[126] Gregor Kiczales. Aspect oriented programming. *ACM Computing Surveys*, 28(154), 1997.

[127] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pages 49–58, New York, NY, USA, 2005. ACM, ACM.

[128] Dongsun Kim and Sooyong Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *SEAMS '09: Proceedings of the International workshop on Software Engineering for Adaptive and Self-Managing Systems in conjunction with ICSE'09*, pages 76–85, Washington, DC, USA, 2009. IEEE Computer Society.

[129] James D. Kiper. Structural testing of rule-based expert systems. *ACM Transactions Software Engineering Methodologies*, 1(2):168–187, 1992.

[130] James G. Kobielus. *BizTalk: Implementing Business-to-Business E-Commerce*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[131] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20, New York, NY, USA, 2006. ACM.

[132] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[133] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990. 101755.

[134] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[135] Jeff Kramer and Jeff Magee. Analysing dynamic change in software architectures: A case study. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 91, Washington, DC, USA, 1998. IEEE Computer Society. 792776 91.

[136] D. R. Kuhn, D. R. Wallace, and Jr. Gallo, A. M. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[137] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IEEE IT Professional*, 10(3):19 – 23, 2008.

[138] Sandeep S. Kulkarni and Karun N. Biyani. Correctness of component-based adaptation. In *CBSE'04: Proceedings of International Symposium on Component-based Software Engineering*, pages 48–58, London, UK. Springer-Verlag.

[139] Bert Lagaisse, Wouter Joosen, and Bart De Win. Managing semantic interference with aspect integration contracts. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software Engineering Properties of Languages for Aspect in conjuction with AOSD'04*, 2004.

[140] Gordon S. lair, Geoff Coulson, Lynne Blair, Mike Clarke, Fabio Costa, Hector Duran, Nikos Parlavantzas, Katia Saikoski, and Anders Andersen. A principled approach to supporting adaptation in distributed mobile environments. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 3–12, Washington, DC, USA, 2000. IEEE Computer Society.

[141] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[142] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[143] Karl Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *Computer Journal of the British Computer Society*, 46(5):542–565, September 2003.

[144] Denivaldo Lopes, Slimane Hammoudi, Jose de Souza, and Alan Bontempo. Metamodel matching: Experiments and comparison. In *ICSEA '06: Proceedings of the International Conference on Software Engineering Advances*, page 2, Washington, DC, USA, 2006. IEEE Computer Society.

[145] Heng Lu, W.K. Chan, and T.H. Tse. Testing pervasive software in the presence of context inconsistency resolution services. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 61–70, New York, NY, USA, 2008. ACM.

[146] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[147] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit, 1987.

[148] A. Markov. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. In R. Howard, editor, *Dynamic Probabilistic Systems (Volume I: Markov Models)*, chapter Appendix B, pages 552–577. John Wiley & Sons, Inc., New York City, 1971.

[149] Nelson Matthys, Sam Michiels, Wouter Joosen, and Pierre Verbaeten. Policy-based management of middleware for distributed sensor applications. In *MiNEMA '08: Proceedings of the 6th workshop on Middleware for network eccentric and mobile applications*, pages 15–17, New York, NY, USA, 2008. ACM.

[150] Nathan McEachen and Roger Alexander. Distributing classes with woven concerns—an exploration of potential fault scenarios. In Peri Tarr, editor, *AOSD'05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 192–200, New York, NY, USA, March 2005. ACM Press.

[151] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[152] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing Verification Reliability*, vol.14(2):105–156, 2004. 1077279.

[153] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[154] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *SEAMS '08: Proceedings of the 2008 international*

workshop on Software engineering for adaptive and self-managing systems in conjuction with ICSE'08, pages 9–16, New York, NY, USA, 2008. ACM.

[155] Brice Morin, Olivier Barais, and Jean-Marc Jézéquel. K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In *MRT'08: Proceedings of the 3rd International Workshop on Models@Runtime in cojunction with MoDELS'08*, Berlin, Heidelberg, October 2008. Springer-Verlag.

[156] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *IEEE Computer*, 42(10):44–51, 2009.

[157] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[158] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 782–796, Berlin, Heidelberg, 2008. Springer-Verlag.

[159] Brice Morin, Gilles Vanwormhoudt, Philippe Lahire, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. Managing variability complexity in aspect-oriented modeling. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 797–812, Berlin, Heidelberg, 2008. Springer-Verlag.

[160] Michael C. Mozer. Lessons from an adaptive home. In *Smart Environments*, pages 271–294, Department of Computer Science and Engineering, The University of Texas at Arlington, Box 19015, Arlington, TX 76019, USA, 2004. Copyright © 2005 John Wiley Sons, Inc.

[161] Hausi Müller, Mauro Pezzè, and Mary Shaw. Visibility of control in adaptive systems. In *ULSSIS '08: Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, pages 23–26, New York, NY, USA, 2008. ACM.

[162] Freddy Munoz and Benoit Baudry. Validation challenges in model composition: The case of adaptive systems. In *ChaMDE 2008: Workshop on Challenges in Model Driven Engineering in conjounction with MODELS'08*, London, UK, September 2008. Springer-Verlag.

[163] Freddy Munoz and Benoit Baudry. A framework for testing model composition engines. In *SC '09: Proceedings of the 8th International Conference on Software Composition*, pages 125–141, Berlin, Heidelberg, 2009. Springer-Verlag.

[164] Freddy Munoz, Benoit Baudry, and Olivier Barais. A classification of invasive patterns in aop. Research report inria-00266555, INRIA, http://hal.inria.fr/inria-00266555/en/, March 2008.

[165] Freddy Munoz, Benoit Baudry, and Olivier Barais. Improving maintenance in aop through an interaction specification framework. In *ICSM'08: 24th International Conference on Software Maintenance, 2008*, pages 77–86, Washington, DC, USA, October 2008. IEEE Computer Society.

[166] Freddy Munoz, Benoit Baudry, Romain Delamare, and Yves Le-Traon. Inquiring the usage of aspect-oriented programming: an empirical study. In *ICSM'09: 25th IEEE International Conference on Software Maintenance*, pages 137–146, Washington, DC, USA, Sep-Oct 2009. IEEE Computer Society.

[167] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[168] Mohamed N. Bennani and Daniel A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 229–240, Washington, DC, USA, 2005. IEEE Computer Society.

[169] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[170] Jürgen Nehmer, Martin Becker, Arthur Karshmer, and Rosemarie Lamm. Living assistance systems: An ambient intelligence approach. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 43–50, New York, NY, USA, May 2006. ACM.

[171] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 54–64, Washington, DC, USA, 2007. IEEE Computer Society.

[172] Cu D. Nguyen, Anna Perini, Paolo Tonella, Simon Miles, Mark Harman, and Michael Luck. Evolutionary testing of autonomous software agents. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 521–528, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[173] Tin A. Nguyen, Walton Perkin, Thomas J. Laffey, and Deanne Pecora. Knowledge base verification. *AI Magazine*, 8(2):69–75, 1987.

[174] Oscar Nierstrasz, Marcus Denker, and Lukas Renggli. Model-centric, context-aware software adaptation. *Software Engineering for Self-Adaptive Systems*, 5525:128–145, 2009.

[175] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[176] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[177] OSGi Alliance. Osgi service platform release 4. [Online]. Available: http://www.osgi.org/Main/HomePage. [Accessed: Jun. 17, 2009], 2007.

[178] George A. Papadopoulos, Wita Wojtkowski, Gregory Wojtkowski, Stanislaw Wrycza, and Jose Zupancic. Applying utility functions to adaptation planning for home automation applications. In *ISD'08: 17th International Conference on Information Systems Development*, Lecture Notes in Computer Science, pages 529–537. Springer-Verlag, August 2008.

[179] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification, and Reliability*, 9(4):263–282, 1999. 10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y.

[180] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac: an aspect-based distributed dynamic framework. *Software Practice Experts*, 34(12):1119–1148, 2004.

[181] Gilles Perrouin, Julien Deantoni, Jean marc Jézéquel, and Franck Chauvel. Modeling the variability space of self-adaptive applications. In *DSPL'08: 2nd Dynamic Software Product Lines Workshop in conjunction with SPLC 2008*, 8 September 2008.

[182] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A component-based and aspect-oriented model for software evolution. *International Journal of Computer Applications in Technology (IJCAT)*, 2008.

[183] Mauro Pezze and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. Number 1. Wiley, John Sons, Incorporated, 2007.

[184] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM.

[185] Sriplakich Prawee, Guillaume Waignier, and Anne-Françoise Le Meur. Enabling Dynamic Co-evolution of Models and Runtime Applications. In *COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1116 – 1121, Washington, DC, USA, 2008. IEEE Computer Society.

[186] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[187] European Union Seventh Framework Programme. Dynamic variability in ccomplex, adaptive systems (diva). http://www.ict-diva.eu/, 2008.

[188] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86, New York, NY, USA, 2008. ACM.

[189] Syed Saif ur Rahman, Nasreddine Aoumeur, and Gunter Saake. An adaptive eca-centric architecture for agile service-based business processes with compliant aspectual .net environment. In *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 240–247, New York, NY, USA, 2008. ACM.

[190] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In *ICMAS-95: Proceedings of the first international conference on Multi-Agent Systems*, pages 312–319, New York, NY, USA, 1995. ACM.

[191] Andreas Rasche and Andreas Polze. Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 164, Washington, DC, USA, 2003. IEEE Computer Society.

[192] Awais Rashid and Ana Moreira. Domain models are NOT aspect free. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS'06: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems,*, volume 4199 of *Lecture Notes in Computer Science*, pages 155–169, Washington, DC, USA, 2006. Springer.

[193] Y. Raghu Reddy. *An approach to composing aspect-oriented design models*. PhD thesis, Fort Collins, CO, USA, 2006. Adviser-Robert France.

[194] Y. Raghu Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development*, 3880:75–105, 2006.

[195] Giovanni Rimassa, Dominic Greenwood, and Martin E. Kernland. The living systems technology suite: An autonomous middleware for autonomic computing. In *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 33, Washington, DC, USA, 2006. IEEE Computer Society.

[196] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *FSE-12: International*

Symposium On Foundations Of Software Engineering, pages 147–158. ACM, October 2004.

[197] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. *Software Engineering for Self-Adaptive Systems*, 5525:164–182, 2009.

[198] Romain Rouvoy, Mikael Beauvois, and Frank Eliassen. Dynamic aspect weaving using a planning-based adaptation middleware. In Rüdiger Kapitza and Hans P. Reiser, editors, *Proceedings of the 2nd Workshop on Middleware-Application Interaction (MAI'08)*, page 1–6, Oslo, Norway, 2008. ACM.

[199] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein O. Hallsteinsen, and Erlend Stav. Composing components and services using a planning-based adaptation middleware. In *SC'08: 7th International Symposium on Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 52–67, Budapest, Hungary, march 2008. Springer.

[200] Michele Sama, David S. Rosenblum, Zhimin Wang, and Sebastian Elbaum. Model-based fault detection in context-aware adaptive applications. In *FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 261–271, New York, NY, USA, 2008. ACM.

[201] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, Washington, DC, USA, 2008. IEEE Computer Society.

[202] Yoshikazu Sawaragi, Hirotaka Nakayama, and Tetsuzo Tanino. *Theory of multiobjective optimization*. Academic Press, Orlando, 1985.

[203] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), February 2006.

[204] G. Seroussi and N.H. Bshouty. Vector sets for exhaustive testing of logic circuits. *Information Theory, IEEE Transactions on*, 34(3):513 –522, may 1988.

[205] Giovanna Di Marzo Serugendo, John Fitzgerald, Alexander Romanovsky, and Nicolas Guelfi. A generic framework for the engineering of self-adaptive and self-organising systems. In Kirstie Bellman, Michael G. Hinchey, Christian Müller-Schloer, Hartmut Schmeck, and Rolf Würtz, editors, *Organic Computing - Controlled Self-organization*, number 08141 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[206] Yoo Shin and Harman Mark. Pareto efficient multi-objective test case selection, 2007. 1273483 140-150.

[207] Hui Song, Yingfei Xiong, Franck Chauvel, Gang Huang, Zhenjiang Hu, and Hong Mei. Generating Synchronization Engines between Running Systems and Their Model-Based Views . In Nelly Bencomo, Gordon Blair, Robert France, Cedric Jeanneret, and Freddy Munoz, editors, *MRT'09: 4th International Workshop Models@run.time in jounction with Models 2009*, volume 509 of *CEUR Workshop Proceedings*, pages 1–10, Berlin, Heidelberg, 10 2009. Springer-Verlag.

[208] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM'05: 21st International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA, September 2005. IEEE Computer Society.

[209] Sun Microsystems. *Java Remote Method Invocation Specification*, November 1996.

[210] Bholonath Surajbali, Geoff Coulson, Phil Greenwood, and Paul Grace. Augmenting reflective middleware with an aspect orientation support layer. In *ARM '07: Proceedings of the 6th international workshop on Adaptive and reflective middleware*, 2007.

[211] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[212] Motoi Suwa, A. C Scott, and Edward H. Shortliffe. An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine.*, 3(4), 1982.

[213] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Plan-directed architectural change for autonomous systems. In *SAVCBS'07: conference on Specification and verification of component-based systems*, pages 15–21, New York, NY, USA, 2007. ACM.

[214] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From Goals to Components: A Combined Approach to Self-Management. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems'08: International workshop on Software engineering for adaptive and self-managing systems in conjunction with ICSE'08*, New York, NY, USA, 2008. ACM.

[215] Clemens A. Szyperski. Emerging component software technologies - a strategic comparison. *Software - Concepts and Tools*, 19(1):2–10, 1998.

[216] Jianbin Tan, George S. Avrunin, and Lori A. Clarke. Managing space for finite-sate verification. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 152–161, New York, NY, USA, 2006. ACM.

[217] Tom Tourwé, Johan Brichau, and Kris Gybels. On the existence of the AOSD-evolution paradox. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT '03: Software engineering Properties of Languages for Aspect Technologies in conjunction with AOSD'03*, March 2003.

[218] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[219] Kevin Twidle, Emil Lupu, Naranker Dulay, and Morris Sloman. Ponder2 - a policy environment for autonomous pervasive systems. In *POLICY '08: Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*, pages 245–246, Washington, DC, USA, 2008. IEEE Computer Society.

[220] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in jasco. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM.

[221] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Incremental Model Synchronization for Efficient Run-time Monitoring . In Nelly Bencomo, Gordon Blair, Robert France, Cedric Jeanneret, and Freddy Munoz, editors, *MRT'09: 4th International Workshop Models@run.time in jounction with Models 2009*, volume 509 of *CEUR Workshop Proceedings*, pages 1–10, Berlin, Heidelberg, 10 2009. Springer-Verlag.

[222] Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien. A Model-Based Framework to Design and Debug Safe Component-Based Autonomic Systems. In Raffaela Mirandola, Ian Gortona, and Christine Hofmeiste, editors, *International Conference on the Quality of Software-Architectures Architectures for Adaptive Software Systems*, volume 5581 of *Lecture Notes in Computer Science*, pages 1–17, Pennsylvania États-Unis d'Amérique, 2009. Springer-Verlag.

[223] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. Automated generation of context-aware tests. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 406–415, Washington, DC, USA, 2007. IEEE Computer Society.

[224] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *In Proceedings of GECCO*, pages 1400–1412. Springer-Verlag, 2004.

[225] Joachim Wegener and Matthias Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[226] Alan W. Williams and Robert L. Probert. Formulation of the interaction test coverage problem as an integer problem. In *In Proceedings of the 14th International Conference on the Testing of Communicating Systems (TestCom 2002*, pages 283–298, 2002.

[227] Eclipse foundation Xerox Park, IBM Corporation. Aspectj. http://www.aspectj.org, 2009.

[228] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[229] Chang Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 336–345, New York, NY, USA, 2005. ACM.

[230] Chang Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 292–301, New York, NY, USA, May 2006. ACM.

[231] Myra B. Cohen Xun Yuan and Atif M. Memon. Gui interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 36:81–95, 2009.

[232] Xin-She Yang. *Introduction to Mathematical Optimization: From Linear Programming to Metaheuristics*. Cambridge International Science Publishing, 2008.

[233] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, vol.32(0098-5589):45–54, 2006.

[234] L. A. Zadeh. Fuzzy logic and approximate reasoning. In *Fuzzy sets, fuzzy logic, and fuzzy systems: selected papers by Lotfi A. Zadeh*, pages 238–259. World Scientific Publishing Co., Inc., 1996. 234395.

[235] John Zeleznikow and Dan Hunter. *Building Intelligent Legal Information Systems (Computer Law, No 13)*. Springer, first edition, December 1994.

[236] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, May 2006. ACM.

[237] Ji Zhang and Betty H.C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361 – 1369, 2006. Architecting Dependable Systems.

[238] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 161–172, New York, NY, USA, March 2009. 2009. ACM.

List of Figures

Listings

Publications

International Conferences

1. Freddy Munoz, Benoit Baudry, Romain Delamare, and Yves Le Traon. **Inquiring the usage of aspect-oriented programming: an empirical study**. In *ICSM09, 25th International conference on Software Maintentance*. September 2009, Edmonton, Alberta, Canada.
2. Freddy Munoz and Benoit Baudry. **A framework for testing model composition engines**. In *SC 2009, 8th international conference on Software Composition*. July 2009, Zurich, Switzerland.
3. Freddy Munoz, Benoit Baudry, and Olivier Barais. **Improving Maintenance in AOP Through an Interaction Specification Framework**. In *ICSM08, 24th International conference on Software Maintentance*. September 2008, Beijing, China.

International Workshops

1. Freddy Munoz, Benoit Baudry, and Olivier Barais. **Vigilant usage of Aspects**. In *Proceedings of ADI 2007 - Workshop on Aspects, Dependencies, and Interactions at ECOOP 2007*. July 2007, Berlin, Germany.
2. Freddy Munoz and Benoit Baudry. **Validation challenges in model composition: The case of adaptive systems**. In *Proceedings of ChaMDE 2000 - Workshop on Challenges in Model Driven Engineering in conjounction with MODELS'08*. September 2008, Toulouse, France.

Other Publications

1. Freddy Munoz, Benoit Baudry, and Olivier Barais. **A classification of invasive patterns in AOP**. In *INRIA Research report 00266555 http://hal.inria.fr/inria-00266555/en/ - (IRISA/INRIA) Institut National de Recherche en Informatique et Automatique, INRIA Bretagne Atlantique*. March 2008, Rennes, France.
2. Freddy Munoz and Benoit Baudry. **Artificial table testing dynamically adaptive systems**. In *INRIA Research report 00365874, http://hal.inria.fr/inria-00365874/en/*

- *(IRISA/INRIA) Institut National de Recherche en Informatique et Automatique, INRIA Bretagne Atlantique.* March 2009, Rennes, France.

Résumé

Les systèmes auto adaptatifs sont des systèmes logiciels capables d'observer leur environnement de travail (par des sondes), raisonner et prendre des décisions sur la façon de s'adapter aux changements environnementaux (par un moteur de raisonnement), et modifier leur structure interne pour prendre les adaptations en compte. Ces systèmes peuvent fournir une aide précieuse dans un grand nombre d'activités humaines. Cependant, ils ne fourniront entièrement leurs promesses que si les ingénieurs peuvent s'assurer que les décisions et les adaptations sont correctes sur toutes les situations. Ceci exige des techniques robustes pour valider que les mécanismes de raisonnement et d'adaptation dans de tels systèmes sont corrects. Dans cette thèse j'adresse la validation des moteurs de raisonnement et des mécanismes d'adaptation par programmation orientée aspect.

Les moteurs de raisonnement sont des éléments logiciels responsables de raisonner sur un grand nombre de facteurs afin de décider comment adapter un système face à des variations dans l'environnement. Il est primordial d'assurer que les décisions prises par ces moteurs sont correctes pour chaque changement d'environnement possible. Une décision erronée peut mener vers une adaptation défectueuse qui peut empêcher le système de fonctionner correctement. Cependant, valider que les moteurs de raisonnement prennent la bonne décision entraine des défis dus au grand nombre de changements environnementaux possibles. Dans cette thèse je présente multi dimesional covering arrays (MDCA) pour échantillonner les conditions environnementales qui peuvent affecter la prise des décisions. MDCA vise spécifiquement les environnements qui peuvent déclencher des décisions complexes en intégrant explicitement la notion de l'histoire dans l'échantillon d'environnement.

La programmation orientée aspect (AOP) fourni les moyens aux ingénieurs d'augmenter ou remplacer la structure ou le comportement du système, ces propriétés font d'AOP un bon mécanisme d'adaptation. Cependant, en utilisant l'AOP il est difficile de (i) prévoir des interactions entre différents aspects et le système de base, (ii) contrôler les endroits où les aspects se tisseront, (iii) assurer que les aspects s'exécuteront sans risque pour l'évolution (des aspects ou du système de base). Ces difficultés entravent la validation et l'adoption de l'AOP en général. Dans cette thèse je présente un framework pour la spécification d'interactions dans des programmes orientée aspects (ABIS), qui permet aux ingénieurs de contrôler les interactions entre les aspects et le système de base en spécifiant les endroits où aspects sont autorisés à se tisser. ABIS permet aux ingénieurs de réduire le temps nécessaire pour diagnostiquer et corriger des problèmes dus aux aspects défectueux.

Abstract

Self-adaptive system are software systems capable of sensing their working environment (through sensors), reason and make decisions on how to adapt facing environmental

changes (through a reasoning engine), and reconfigure their internal structure in order apply adaptations (through an adaptation mechanism). These systems can provide effective assistance in a large number of human activities. Yet, they will fully deliver their promises only if system engineers can ensure that decisions and adaptations are correct on all situations. This requires robust techniques for validating that the reasoning process and adaptation mechanisms implemented in such systems are correct. In this thesis I address the validation of reasoning engines and adaptation mechanism base on aspect-oriented programming.

Reasoning engines are pieces of software entitled of reasoning on a large number of factors in order to decide how to adapt given an environmental change. It is critical to ensure that the decisions of these engines are correct for every possible environment change. Wrong decision may lead to faulty adaptations that may prevent the system to work properly. Yet, validating that reasoning engines make the right decision is challenging because of the large number of possible environmental changes. In this thesis I introduce multi-dimensional covering arrays (MDCA) for sampling the environmental conditions that may influence the decision making process. MDCA specifically targets environments that can trigger complex decision by explicitly integrating the notion of history in the environment sample.

Aspect oriented programming (AOP) provide the means to developers to augment or replace the system structure or behavior, properties that make AOP a good adaptation mechanism. Yet, when using AOP it is difficult to (i) foresee interactions between different aspects and the base system, (ii) control the places that aspects will advise and invade, (iii) ensure that aspects will perform safely on evolution (aspects or the base system). These difficulties hamper the validation and adoption of AOP in general. In this thesis I introduce an interactions specification framework (ABIS), which allows developers to control the aspects interactions with the base system by specifying the places that aspects are allowed to advice. ABIS enables developers to reduce the time needed to diagnose and correct problems due to faulty aspects.

VU :

Le Directeur de Thèse
Jean-Marc Jézéquel

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
Olivier Ridoux