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

École doctorale Matisse

présentée par

Brice Morin

préparée à l'IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
Composante Universitaire : IFSIC

Leveraging Models from Design-time to Runtime to Support Dynamic Variability

Thèse soutenue à Rennes
Le 17 Septembre 2010

devant le jury composé de :

Thomas JENSEN
Directeur de Recherche CNRS
Président

Betty H.C. CHENG
Professeur Michigan State University
Rapporteur

Valérie ISSARNY
Directeur de Recherche INRIA Paris-Rocquencourt
Rapporteur

Jean-Bernard STEFANI
Directeur de Recherche INRIA Rhone-Alpes
Examinateur

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

Olivier BARAIS
Maitre de Conférence à l'Université de Rennes 1
Encadrant

Contents

Acknowledgments

This thesis is the result of three years of work conducted within the Triskell group, probably one of the best team in the world both in terms of work and ambiance (especially during the coffee breaks, the "BBQ chez Greg" and other events).

First of all, I would like to thank Jean-Marc Jézéquel. He has been a great advisor during these 3 years: he kept my focused on my subject and also left me some degrees of freedom to work on (not always so) related topics. This efficient advising allowed me to progressively discover *the good*[1], *the bad*[2] *and the ugly*[3] aspects of the life of a researcher. Obviously, I want to continue in this way.

Another important person I would like to thank is Olivier Barais (*a.k.a.* the Jedi Master of the Bytecode). Even if he is always very busy, he always find the time to answer some questions or to install (hack) prototypes on my computer. I remember that the week we spent in Oslo at SINTEF was particularly productive.

I would also like to thank the other members of the Triskell group. In particular, Grégory Nain and François Fouquet for using my tools and helping me to improve them, for all the technical advices, and also for all the good moments after work.

Of course, I would like to acknowledge the members of the jury for their encouraging comments about my work and the interesting questions they asked. It was a real honor for me to defend my thesis in front of such a challenging jury.

Another group that was (and is still) important for me during these three years is DiVA, the European project in which I was involved during these three years. I would like to thank the guys from Thales and CAS for their important feedback on the tools and the related approach. I am grateful Awais Rashid and Gordon Blair for hosting me at Lancaster University during 3 month and to Nelly Bencomo for all the nice discussions we had.

As a conclusion to these acknowledgments (more precisely, as a transition) I would like to thank the people from SINTEF involved in DiVA. Especially, I am grateful to Arnor Solberg for hiring me as a permanent researcher at SINTEF right after my defense. I am sure I will enjoy his group as much as I enjoyed working for Triskell. Last but not least,

---

[1]many

[2]not that much

[3]very few

I would like to thank Franck Fleurey. His tips about Kermeta and the discussion we had about AOM during my Master internship and at the beginning of my Ph.D. thesis (when he was still working in the Triskell group) were really helpful. Then, when DiVA started (when he joined SINTEF) I really enjoyed collaborating with him on the great topic of MDE for adaptive systems.

Résumé en français

1.1 Context

La société d'aujourd'hui dépend de plus en plus des systèmes logiciels [16] déployés dans de grandes compagnies, banques, aéroports, opérateurs de télécommunication, etc. Ces systèmes doivent être disponible 24H/24 et 7j/7 pour de très longues périodes. Ainsi, le système doit être capable de s'adapter à différents contextes d'exécution, sans interruption (ou très localisé dans le temps et dans l'espace), et sans intervention humaine. Pour pouvoir s'exécuter pendant une longue période, le système doit être ouvert à l'évolution. Il est en effet impossible de prévoir ce que seront les besoins des utilisateurs dans 10 ans.

Une approche prometteuse consiste à concevoir et à implémenter ces systèmes critiques comme des systèmes adaptatifs (DAS, Dynamically Adaptive Systems), qui peuvent s'adapter selon leurs contextes d'exécution, et évoluer selon les exigences utilisateur. Il y a plus d'une décennie, Peyman Oreizy et al. [118] ont défini l'évolution comme "l'application cohérente de changements au cours du temps", et l'adaptation comme "le cycle de monitoring du context, de planification et de déploiement en réponse aux changements de context". Les DASs ont des natures très différentes :

- systèmes embarqués [75] ou systèmes de systèmes [22, 63],

- systèmes purement auto-adaptatifs ou systèmes dont l'adaptation est choisie par un être humain

Quelque soit son type, l'exécution d'un DAS peut être abstrait comme une machine à états [13, 165], où :

- **Les états** représentent les différentes configurations (ou les modes) possibles du système adaptatif. Une configuration peut être vue comme programme "normal", qui fournit des services, manipule des données, exécute des algorithmes, etc.

- **Les transitions** représentent toutes les différentes migrations possibles d'une configuration vers une autre. Ces transitions sont associées à des prédicats sur le contexte et/ou des préférences utilisateur, qui indiquent quand le système adaptatif doit migrer d'une configuration à une autre.

Fondamentalement, cette machine à états décrit le cycle d'adaptation du DAS. L'évolution d'un DAS consiste conceptuellement à mettre à jour cette machine à états, en ajoutant et/ou enlevant des états (configurations) et/ou des transitions (reconfigurations).

L'énumération de toutes les configurations possibles et des chemins de migration reste possible pour de petits systèmes adaptatifs. Ce design en extension permet de simuler et de valider toutes les configurations et reconfigurations possibles, au moment du design [165], et de générer l'intégralité du code lié à la logique d'adaptation du DAS.

Cependant, dans le cas de systèmes adaptatifs plus complexes, le nombre d'états et de transitions à spécifier explose rapidement [105, 51] : le nombre des configurations explose d'une manière combinatoire par rapport aux nombres de features dynamiques que le système propose, et le nombre de transitions est quadratique par rapport au nombre de configurations. Concevoir et mettre en application la machine à états dirigeant l'exécution d'un système adaptatif complexe (avec des millions ou même des milliards de configurations possibles) est une tache difficile et particulèrement propice aux erreurs.

1.2   Challenges Liés à l'Ingénierie des Systèmes Adaptifs Complexes

Betty Cheng et al. ont identifié plusieurs défis liés aux DAS [35], spécifiques à différentes activités : modélisation des différentes dimensions d'un DAS, expression des besoins, ingénierie (design, architecture, vérification, validation, etc) et assurance logicielle. La plupart de ces défis peuvent se résumer à trouver le niveau juste de l'abstraction :

- assez abstrait pour pouvoir raisonner efficacement et exécuter des activités de validation, sans devoir considérer tous les détails de la réalité,

- et suffisamment détaillé pour établir le lien (dans les 2 directions) entre l'abstraction et la réalité c.-à-d., pour établir un lien causal entre un modèle et le système en cours d'exécution.

L'abstraction est l'une des clefs pour maitriser la complexité des logiciels. La clef pour maitriser les systèmes dynamiquement adaptatifs [105, 104] est de réduire le nombre d'artefacts qu'un concepteur doit spécifier pour décrire et exécuter un tel système, et d'élever le niveau d'abstraction de ces artefacts. Selon Jeff Rothenberg [131],

> *Modéliser, au sens large, est l'utilisation rentable d'une chose au lieu d'une autre pour un certain but cognitif. Cela permet d'employer quelque chose qui est plus simple, plus sûre ou meilleur marché que la réalité, pour un but donné. Un modèle représente la réalité pour un but donné ; c'est une abstraction de la réalité dans le sens qu'il ne peut pas représenter tous les aspects de réalité. Cela permet d'envisager le monde d'une façon simplifiée, en évitant la complexité, le danger et l'irrévocabilité de la réalité.*

Dans les DAS complexes, comme ceux adressés dans le projet DiVA [147], deux aspects importants de la réalité que nous voulons abstraire sont le DAS lui-même et son contexte d'exécution. Ceci soulève les questions suivantes :

1. Comment abstraire le contexte d'exécution en un modèle de haut niveau pouvant servir de base au raisonnement et à la prise de décision ?

2. Comment raisonner efficacement sur le contexte courant afin de trouver un ensemble de features bien adaptées, sans devoir énumérer tous les contextes possibles ou de nombreuses règles d'adaptation ?

3. Comment éviter que le DAS oscille continuellement quand le contexte oscille légèrement autour de seuils critiques ?

4. Comment établir un lien entre la configuration (architecture) correspondant à un ensemble de features, sans devoir spécifier toutes les configurations possibles à l'avance, tout en préservant un degré élevé de validation ?

5. Comment faire migrer un DAS de sa configuration courante vers une nouvelle configuration sans devoir écrire à la main tous les scripts de bas niveau et spécifiques à une plateforme d'exécution donnée.

Ces questions visent à élever le niveau d'abstraction, et à réduire le nombre d'artefacts requis pour spécifier et exécuter des DAS, tout en préservant un degré élevé de validation, d'automatisation et d'indépendance par rapport à des choix technologiques (par exemple, outils utilisés au design, plateformes d'exécution, système de règle pour diriger l'adaptation dynamique, etc.).

1.2.1   Vers du design continu

Au 6ème siècle avant J.C., Sun Tzu a mis en évidence le rôle de la réflexion dans *l'art de la guerre* [154] :

> *Qui connaît son ennemi et se connaît lui-même, peut livrer cent batailles sans jamais être en péril.*
> *Qui ne connaît pas son ennemi mais se connaît lui-même, pour chaque victoire, connaîtra une défaite.*
> *Qui ne connaît ni son ennemi ni lui-même, perdra inéluctablement toutes les batailles.*

Cette citation ne concerne évidemment pas les DAS. Cependant, la réflexion [23, 94] est un concept bien établi dans le contexte de l'adaptation dynamique. Cette citation peut être transposée au champ lexical des DAS. Ici, "lui-même" serait le système adaptatif lui-même, et "l'ennemi" serait le contexte d'exécution dans lequel le système évolue. Finalement, les "batailles" seraient les reconfigurations dynamiques du système. Avec

une connaissance précise du contexte et du système lui-même, il est possible de prendre de bonnes décisions et de faire migrer le DAS de façon fiable, en fonction du contexte courant.

Plus récemment, Daniel G. Bobrow et al. [23] ont défini la réflexion comme :

> *La capacité d'un programme à manipuler comme données quelque chose représentant l'état du programme lui-même pendant sa propre exécution. Il y a deux aspects d'une telle manipulation : introspection et intercession. L'introspection est la capacité d'un programme d'observer et donc raisonner au sujet de son propre état. L'intercession est la capacité d'un programme à modifier son propre état d'exécution ou de changer son interprétation.*

Cela consiste fondamentalement à maintenir un lien causal entre la réalité (c.-à-d. le system en cours d'exécution) et un modèle de la réalité. Un changement significatif du système en cours d'exécution met à jour le modèle, et n'importe quelle modification significative du modèle affectera le système en cours d'exécution. Cela contredit la citation de Jeff Rothenberg puisque ce modèle de réflexion n'évite pas le danger et l'irrévocabilité, en raison de la synchronisation forte entre la réalité et le modèle.

Nous admettons que la réflexion est un concept fondamental pour mettre en oeuvre les systèmes adaptatifs, mais nous proposons d'aller plus loin dans cette thèse afin d'y inclure les arguments de Jeff Rothenberg. Le modèle doit toujours refléter ce qui arrive dans le système courant (introspection), pareillement à un miroir, de sorte qu'il soit possible de toujours raisonner sur un modèle à jour. Cependant, il faut offrir plus de liberté pour manipuler, transformer, valider, etc. le modèle sans modifier directement le système en cours d'exécution. C'est le principe de base de l'intelligence humaine (et de l'intelligence artificielle [135] dans une certaine mesure). Un humain peut établir mentalement plusieurs modèles potentiels de la réalité et évaluer mentalement ces modèles au moyen de scénarios [18][4] : que se produirait-il si je faisais cette action ? Pendant ce raisonnement mental, la manipulation du modèle n'affecte pas la réalité, tant qu'une solution acceptable n'ait été trouvée. Enfin, cette solution est effectivement mise en oeuvre, ce qui impacte la réalité. En d'autres termes, le modèle mental est re-synchronisé avec la réalité. Dans le cas où un aspect de la réalité change pendant le processus de raisonnement, le modèle est mis à jour et le processus de raisonnement reconstruit des modèles mentaux, idéalement en mettant à jour les modèles déjà existants.

Un autre aspect de la réalité qu'on souhaite abstraire est le contexte d'exécution dans lequel le système évolue. Cependant le fossé entre l'espace de conception (design) et l'espace d'exécution (runtime) est assez large. En effet, les sondes intégrés au runtime produisent des flots (quasi) continus de valeurs brutes, avec très peu d'abstraction. Au design, les concepteurs utilisent généralement des valeurs qualitatives [33, 51], comme *haut*, *moyen* ou *bas*, pour décrire l'environnement d'un DAS et sa logique d'adaptation.

---

[4]Voir l'encart par Bran Selic.

1.2.2 Gestion de la Variabilité, Dérivation de Configurations

Comme nous l'avons déjà mentionné, spécifier et mettre en oeuvre explicitement la machine à états dirigeant l'exécution d'un DAS devient rapidement difficile en raison de l'explosion combinatoire du nombre de configurations, et de l'explosion quadratique du nombre de transitions.

Tandis qu'il est encore possible de définir cette machine à états pour des petits systèmes adaptatifs [14, 165], cela devient presque impossible pour des systèmes adaptatifs complexes, avec des millions ou même des milliards de configurations [51, 105]. Même si les états et les transitions sont spécifiés à un haut niveau d'abstraction, il devient rapidement difficile pour un humain, et même pour une machine, de définir un nombre si élevé d'artefacts [51, 105].

Récemment, Sven Hallsteinsen et al. ont conceptualisé les DAS comme des lignes de produits dynamiques [70, 69] (DSPL, Dynamic Software Product Lines) dans lesquelles la variabilité est fixé jusqu'au runtime. Semblables aux lignes de produits traditionnelles [38] (SPLs), l'idée fondamentale est de capitaliser sur les parties communes et de contrôler précisément les variations entre les différents produits. Ainsi, le nombre élevé de configurations possible pour un DAS est décrit en intention, plutôt qu'en extension.

Dans une SPL, les produits sont la plupart du temps dérivés selon des décisions humaines, de l'expression des besoins (requirement) au déploiement [66]. En revanche, le processus de dérivation d'une DSPL est plus complexe et plus varié. À la différence d'une SPL "classique", les produits d'une DSPL sont fortement dépendants : le système doit commuter dynamiquement et sans risque de sa configuration courante vers une autre configuration. La décision de dériver un produit (c.-à-d. s'adapter vers une autre configuration) dépend du type de système adaptatif :

- dans les systèmes Auto-Adaptatifs (SAS, Self-Adaptive Systems), par exemple des systèmes adaptatifs embarqués dans des avions [118], le processus de dérivation est entièrement automatisés et suit souvent la boucle autonomique MAPE (Monitor-Analyze-Plan-Execute) [81].

- dans les systèmes dont l'adaptation est dirigée par un être humain, par exemple un système domotique [116], le choix des features à intégrer est la plupart du temps dirigé par l'utilisateur final.

La communauté SPL [11, 38, 169] propose déjà des formalismes et des notations, telles que les feature diagrams, pour décrire une gamme de produits en intention plutôt qu'en extension. L'idée fondamentale est décrire les points communs entre les produits et d'offrir les constructions néc´essaires (alternatives, options, choix de n parmi p, etc.) et des contraintes (exclusions mutuelles, dépendances, etc.) pour décrire correctement la variabilité parmi les produits. De cette façon, les concepteurs n'ont pas besoin d'énumérer toutes les configurations possibles. Cependant, la décomposition d'un système en points com-

muns et points de variation nécessite des mécanismes efficaces et expressifs de composition pour pouvoir automatiquement dériver des configurations à partir de sélections de features.

La communauté software composition (SC) (qui inclue notamment les communautés AOSD (Aspect-Oriented Software Development), FOSD (Feature-Oriented Software Development) et CBSD (Component-Based Software Development)) propose des mécanismes de compositions différents mais complémentaires, tels que le tissage d'aspects, les mixins, la composition de features ou de composants, etc. Récemment, beaucoup d'approches proposent d'appliquer de tels méchanismes de composition dans le cadre des SPL "classiques" : niveau code [80, 5, 6, 48, 97] ou niveau model [88, 77, 64, 102, 100, 120].

1.2.3   Rendre la boucle d'adaptation explicite

La séparation des préoccupations [119] est une pratique bien établie de génie logiciel. Dans les systèmes adaptatifs, la logique d'adaptation doit être séparée de la logique métiers [12, 10, 14, 40, 52, 122, 123, 133, 134, 165, 168], pour améliorer la compréhension, la maintenance, la testabilité et la modularité de ces systèmes. Au delà de la séparation entre logique d'adaptation et logique métier, il est important de séparer clairement les composants réalisant la logique métier des composants responsables de l'adaptation dynamique. Selon Betty Cheng et al. [35] :

> *Comprendre et raisonner sur les boucles de contrôle des systèmes auto-adaptatifs*
> *est un pré-requis permettant de faire évoluer la technologie des systèmes auto-adaptatifs*
> *d'une approche ad-hoc, "essai-échec" vers une approche disciplinée.*

En considérant les boucles de controle comme des entités de première ordre, il possible de raisonner sur leurs architectures, leurs propriétés (stabilité, réactivité, exécutions, etc.), et de configurer (voir même reconfigurer dynamiquement) ces boucles de contrôle selon les besoins.

1.2.4   Réflex et réflexion : vers des systèmes adaptatifs inspirés par l'être humain

La plupart des approches d'ingénierie pour les systèmes adaptatifs proposent seulement un paradigme pour exprimer la logique d'adaptation. Le paradigme de raisonnement le plus commun est ECA (Event-Condition-Action) [40, 41] qui consiste fondamentalement à mapper des fragments de contextes à des actions à éxécuter (scripts de reconfiguration). Par exemple, si un feu est détecté, des sprinklers doivent être activés. Comme relevé dans [51], il est facile de comprendre et mettre en application chaque règle individuellement : il s'agit simplement d'un ensemble de blocs *if-then*. Cependant, définir la logique d'adaptation uniquement à l'aide de règle ECA nécéssite de définir de nombreuses règles qu'il est parfois difficile de comprendre dans leur ensemble [51]. En effet, un contexte

spécifique peut déclencher plusieurs règles (puisque les conditions sont définies sur des fragments de contexte), qui peuvent être en conflit. Il faut ainsi définir des contraintes additionnelles (priorités, exclusions, etc.) pour contrôler correctement l'ensemble de ces règles ECA.

Une autre manière commune d'exprimer la logique d'adaptation est de définir des buts que le système doit optimiser [51, 62, 34]. Pour chaque feature du système, le concepteur doit préciser comment elle affecte des propriétés de QoS. Il faut également indiquer quelles propriétés doivent être optimiser dans quels contextes. Au runtime, la boucle de contrôle doit trouver le meilleur choix de features, qui optimise au mieux les propriétés qui sont importantes dans le contexte courant. Comme noté dans [51] l'utilisation de buts est plutôt intuitive et simple. En effet, le choix des features à intégrer est laissé au système lui-même. Cependant, les algorithmes d'optimisation multidimensionnelles nécessitent souvent beacoup de ressources et/ou de temps.

Par analogie, le corps humain a deux méchanismes de "raisonnement" : réflex[5] et réflexion[6].

Les règles ECA peuvent être vues comme des réflexes, puisqu'elles n'impliquent pas vraiment de raisonnement. Dans le cas où le système est dans un contexte critique, ces règles peuvent rapidement modifier le système dans une configuration acceptable. Les techniques à base de buts matchent parfaitement la définition de "pensée". Selon les ressources ou le laps de temps accordé au raisonnement, le système peut évaluer différentes configurations et trouver celle qui offre le meilleur compromis.

Les règles ECA et les règles à base de buts ont des avantages et des inconvénients complémentaires. D'une part, des règles d'ECA peuvent être traitées efficacement au runtime. Cependant, il devient rapidement difficile de spécifier un système adaptatif en utilisant seulement des règles ECA. D'autre part, les règles à base de buts permettent de spécifier la totalité de la logique d'adaptation à un haut niveau d'abstraction mais impliquent souvent un traitement plus lourd au runtime. Plutôt que de se limiter au choix d'un paradigme, nous pensons que la combinaison de plusieurs de ces paradigme de raisonnement permet de tirer parti de leurs avantages respectifs tout en limitant leurs inconvénients.

1.2.5 Réactivité Versus Stabilité

Un des critères les plus évidents d'un système adaptatif est sa capacité de s'adapter selon des stimuli externes (contexte d'exécution, préférences d'utilisateur, etc.). Cependant,

---

[5]d'après Wikipédia (FR) : Un réflexe est une réponse musculaire involontaire, stéréotypée et très rapide à un stimulus. Une activité réflexe est induite par un arc réflexe, le mécanisme de réponse intégrée d'un centre nerveux sans intervention du cerveau et de la volonté consciente. Les réflexes sont souvent des réactions de défense, comme le retrait du membre en cas de brûlure, avant que le cerveau ait perçu la douleur.

[6]d'après Wikipédia (EN, traduit) : Penser permet à un être humain de modéliser le monde afin d'agir en fonction de ses objectifs, plans ou désirs.

l'adaptation dynamique n'est pas immédiate et dérange inévitablement le système en cours d'exécution, puisque quelques composants doivent parfois être arrêtés et redémarrés pour pouvoir commuter sans risque d'une configuration à une autre.

Le contexte dans lequel un système adaptatif évolue est très dynamique et peut potentiellement changer plus rapidement que le système adaptatif lui-même. Les propriétés de QoS telles que la bande passante, le CPU, la mémoire, etc peuvent varier très rapidement et dépendent de beaucoup de paramètres, la plupart d'entre eux étant hors de contrôle (du système adaptatif) : le nombre de systèmes qui utilisent le réseau, la taille et la quantité de données échangées, qualité du réseau, etc. Dans le pire cas (où une variable de contexte oscille autour d'un seuil qui déclenche une reconfiguration) le système adaptatif peut osciller sans interruption entre deux configurations. Encore pire, si les fluctuations du contexte sont plus rapides que l'adaptation, le système adaptatif pourrait continuellement se retrouver dans une configuration qui n'est pas adaptée au contexte courant. Dans ce cas-ci, il serait préférable de commuter dans un mode neutre par rapport aux variables qui oscillent.

Mettre en oeuvre un compromis entre réactivité et stabilité n'est cependant pas simple. L'amélioration de la stabilité peut être réalisée en utilisant pour des fenêtres de temps, des cycles d'hystérésis [155], de la logique floue [87] ou du traitement d'événements complexes (CEP, Complex Event Processing) [93] etc. De telles techniques peuvent cacher des fluctuations de contexte, offrant ainsi une base plus stable pour le raisonnement. Cependant, cela pourrait également cacher des contextes critiques où le système doit s'adapter. Pour privilégier une réactivité acceptable, il est souvent nécessaire de travailler à un niveau plus bas d'abstraction (seuil dur sur des valeurs directement fournies par les sondes), avec peu d'analyse, pour pouvoir réagir rapidement selon le contexte.

1.3   Contributions

Dans cette thèse, nous proposons de tirer parti des dernières avancées en Ingénierie des Modèles (MDE, Model-Driven Engineering [139]) aussi bien pendant la conception (design) qu'à l'exécution (runtime) [18]. Les aspects fondamentaux d'un DAS (sa variabilité (dynamique), son contexte, sa logique d'adaptation et son architecture) à l'aide de métamodèles dédiés. En opérant à un niveau élevé de l'abstraction, il est ainsi possible de raisonner efficacement, en cachant les détails non pertinents, et d'automatiser le processus de reconfiguration dynamique. Dans cette thèse, nous contribuerons en particulier sur les questions suivantes :

1. Comment construire la configuration (architecture) correspondant à un ensemble de features, sans devoir spécifier toutes les configurations possibles à l'avance, tout en préservant un degré élevé de validation ?

2. Comment reconfigurer automatiquement le DAS de sa configuration courante à une

configuration nouvellement produite, sans devoir écrire à la main des scripts de reconfiguration de bas niveau et spécifiques à la plateforme d'exécution ?

1.3.1 Modélisation des Systèmes Adaptatifs

Dans le contexte du projet européen DiVA, et plus particulièrement en collaboration avec Franck Fleurey, nous avons modelé quatre dimensions/aspects d'un système adaptatif [50] :

- sa variabilité, qui décrit les différentes features du système, et leurs natures (options, alternatives, etc.)

- son environnement/contexte, qui décrit les points important du contexte nous voulons surveiller (environnement), aussi bien que le contexte courant.

- sa logique d'adaptation, qui décrit quand le système doit s'adapter. Cela consiste à définir quelles features (du modèle de variabilité) choisir en fonction du contexte courant.

- son architecture, qui décrit la configuration du système courant en termes de concepts architecturaux.

Cette contribution est présentée dans [50, 104] et détaillée dans le Chapitre 3 de ce document.

1.3.2 Modélisation par Aspects pour la Dérivation de Configurations

La communauté SPL [11, 38, 169] propose déjà des formalismes et des notations bien établis, tels que des feature diagrams, pour contrôler la variabilité en fond décrivant une gamme de produits sans devoir énumérer tous les produits individuellement. La communauté SC propose un éventail de techniques différentes mais complémentaires, qui peuvent être utilisées pour dériver des produits depuis un modèle de ligne de produits [80, 120, 106]. Nous nous appuyons sur cette expérience pour décrire la variabilité de DASs, ou DSPLs, et dériver des configurations depuis un modèle de variabilité. Puisque nous utilisons intensivement des modèles, nous proposons naturellement d'utiliser la Modélisation par Aspects (AOM, Aspect-Oriented Modeling) afin de raffiner et composer les features [120, 64, 160]. L'AOM se base sur des approches formelles, telle que la théorie des graphes [24], ce qui permet de valider tôt dans le cycle de dévelopement, sans devoir énumérer toutes les configurations possibles (combinaisons de features). Alors que la littérature est très prolifique au sujet des approches AOM [9, 88, 102, 78, 64], peu d'implémentations sont réellement disponibles. Dans cette thèse, nous présentons comment nous étendons l'approche SmartAdapters [88, 101], que nous avons développé au sein de l'équipe INRIA Triskell. Cependant, notre approche n'est pas spécifique à ce tisseur d'aspects. Il est en effet possible d'utiliser d'autres tisseurs, voire même de simples transformations de modèles pour produire des configurations.

Cette contribution est présentée dans [106, 105, 104] et détaillée dans le Chapitre 4 de ce document.

1.3.3   (Dé)coupler le Model de Réfléction (d')avec la Réalité

Nous avons précédemment fait une analogie entre les systèmes adaptatifs et l'intelligence humaine, et sa capacité d'établir (si nécessaire) des modèles mentaux reflétant la réalité, mais qui ne sont pas directement couplés à la réalité. Cette thèse n'explorera pas plus loin cette analogie. Cependant, nous pensons qu'un model de réflexion indépendant de la réalité est une avancée importante que les systèmes adaptatifs modernes doivent intégrer. Les DAS modernes deviennent tellement complexes qu'il devient difficile de produire, valider, simuler, etc. toutes configurations possibles d'un DAS lors du design. Il est également inconcevable dans beaucoup de domaines (santé, systèmes critiques, etc.) de faire migrer un système vers une configuration qui n'a pas été précédemment validée, alors que les vies humaines dépendent directement de ces systèmes logiciels.

Nous détaillerons comment les techniques MDE permettent de construire des configurations qui n'affecte pas la réalité, et comment synchroniser automatiquement ces configurations (une fois validées) avec le système courant, sans devoir écrire à la main des scripts de reconfiguration de bas niveau et spécifiques à la plateforme d'exécution. Si une configuration n'est pas valide, elle est simplement rejetée : il n'y a pas besoin d'effectuer un roll-back sur le système, puisqu'il n'a pas été affecté par la construction de ce modèle invalide.

La faculté d'un DAS à établir des modèles qui ne sont pas directement liés à la réalité permet de valider chaque configuration seulement lorsque c'est nécessaire. Cette validation en ligne fournit un haut-degré de confiance dans le système adaptatif : le processus d'adaptation ne fera jamais migrer le système vers une configuration qui n'a pas été validée. Il est important de noter que cette thèse ne vise pas à contribuer sur le processus de validation lui-même. Au lieu de cela, le fait de pouvoir découpler et en synchroniser le modèle de réflexion/avec la réalité permet d'appliquer des techniques de validations existantes et même d'intégrer les futures avancées dans le domaine de la validation.

Cette contribution est présentée dans [106, 105, 104, 103, 107, 111] et détaillée dans le Chapitre 5 de ce document.

1.3.4   Mise en Oeuvre et Validation

Les différentes contributions de cette thèse ont été effectivement implémentées et intégrées dans une architecture de référence. Les différentes tâches liées à la configuration et à la reconfiguration d'un DAS ont été encapsulées dans des composants. Ces composants échangent principalement des modèles, qui décrivent les différentes features du système ainsi que son contexte d'exécution. Nous tirons parti de ces modèles pour automatiser entièrement le déploiement initial et la reconfiguration des systèmes adaptat-

ifs [105, 106, 104]. À tout moment, il est possible de modifier les modèles définissant le DAS (design continu). Cette architecture de référence a été utilisée et démontrée avec succès dans plusieurs contextes :

- Plusieurs (vraies) démonstrations ont été présentées au workshop Models@Run.Time [114, 111, 103] et à la conférence AOSD [107].

- Elle a été intégrée comme moteur de configuration et de reconfiguration de la plate-forme domotique EnTiMid développé par l'équipe Triskell, principalement par Grégory Nain.

- Elle est actuellement utilisée par les partenaires industriels du projet européen DiVA pour réaliser leurs études de cas : Un système de gestion de crise dans un aéroport (Thales), et un système de gestion de relations client (CAS Software AG).

- Puisque l'architecture de référence est elle-même un système à base de composants, il est possible de bootstrapper c.-à-d., utiliser cette architecture de référence pour déployer (et même pour adapter dynamiquement) une autre instance de cette architecture. En d'autres termes, il devient possible d'adapter dynamiquement le processus d'adaptation. Cette idée sera discutée comme l'une des perspectives de cette thèse (Section 9.2).

L'architecture de référence est présentée dans [104] et détaillée dans le Chapitre 6 de ce document. Le Chapitre 7 (dédié à la validation) applique notre approche sur 2 études de cas réalistes.

Part I

Introduction & State-of-the-Art

Chapter 1

Introduction

Contents

1.1 Context

Today's society increasingly depends on software systems deployed in large companies, banks, airports, telecommunication operators, and so on. These systems must be available 24/7 for very long period [16]. To sustain the availability, the system should be able to adapt to different execution contexts, with no (or very localized in time and in space) interruption, and with no human intervention. To be able to run for a long period, the

system should be open to evolution, because it is impossible to predict what the user requirements will be in 10 years.

A promising approach is to implement such critical systems as Dynamically Adaptive Systems (DASs), which are able to adapt according to their execution context, and even evolve according to changing user requirements. More than a decade ago[1] Peyman Oreizy and colleagues [118] defined evolution as "*the consistent application of change over time*", and adaptation as "*the cycle of detecting changing circumstances and planning and deploying responsive modifications*". DASs have very different natures, and can range from:

- small embedded systems [75] to large systems of systems [22, 63], or from

- human-driven [116] to purely self-adaptive systems [81, 82, 8]

Whatever its type, the execution of a DAS can be abstracted as a state machine [13, 165], where:

- **States** represent the different configurations (or modes) of the adaptive system. A configuration can be seen as a "normal" program, which provides services, manipulates data, executes algorithms, etc.

- **Transitions** represent all the different possible migration paths from one configuration to another. Transitions are associated with conditions on the context and/or user preferences, which specify when the adaptive system should migrate from one configuration to another.

Basically, this state machine drives the adaptation cycle of the DAS. Evolving the DAS means evolving this state machine, by adding and/or removing states (configurations) and/or transitions (migration paths).

The enumeration of all the possible configurations and migration paths of a small adaptive system is still feasible. It allows performing extensive simulation and validation at design-time [165] and generating the code related to the adaptation logic of the DAS. However, when the adaptive system becomes more complex, the number of states and transitions to specify and to implement rapidly explodes [105, 51]. Indeed, the number of configurations explodes in a combinatorial way *w.r.t.* the number of variable features the system propose, and the number of transitions is quadratic *w.r.t.* the number of configurations. Fully specifying and implementing the state machine driving the execution of an adaptive system rapidly becomes a daunting and error-prone task in the context of complex DAS, with million or even billion possible configurations.

---

[1]In 1999.

1.2 Challenges Related to the Engineering of Dynamically Adaptive Systems

Betty Cheng and colleagues identified several challenges of engineering DAS [35], related to different activities: modeling dimensions, requirements, engineering (modeling, architecture & design, middleware support, verification & validation) and assurance. Most of the identified challenges can be summarized as: finding the right level of abstraction, which

- is abstract enough to be able to efficiently reason, perform validation, with no need to consider all the details of the reality

- is detailed enough to bridge the gap (in both directions) between abstraction and reality *i.e.*, to establish a causal connection between a model and a running system.

Abstraction is one of the keys to deal with complexity. The key to tame dynamically adaptive systems [105, 104] is to reduce the number of artifacts a designer has to specify to describe and execute an adaptive system, and to raise the level of abstraction of these artifacts. According to Jeff Rothenberg [131],

> *Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is **simpler, safer or cheaper than reality** instead of reality for some purpose. A model represents reality **for the given purpose**; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, **avoiding the complexity, danger and irreversibility of reality**.*

In complex dynamically adaptive systems, such as the ones addressed in the DiVA project [147], two important aspects of the reality we want to abstract are the dynamically adaptive system itself and the execution context in which the system evolves. This raises the following questions:

1. How to abstract the actual execution context into a high-level model, which offers a solid basis for reasoning and decision making, and hides irrelevant details?

2. How to efficiently reason on the current context in order to find a well-suited set of features, with no need to enumerate all the possible contexts or to enumerate numerous rules?

3. How to avoid the adaptive system to continuously oscillates when the context slightly oscillates around critical thresholds?

4. How to actually build the configuration (architecture) corresponding to a set of features, with no need to fully specify all the possible configuration beforehand, while preserving a high degree of validation?

5. How to plan and executes the changes from the current configuration to a newly produced configuration, with no need to write by hand low-level platform-specific reconfiguration scripts?

These questions aim at raising the level of abstraction, and reducing the number of the artifacts needed to specify and execute dynamically adaptive systems, while preserving a high degree of validation, automation and independence *w.r.t.* specific technological choices (*e.g.*, design-time tools, runtime execution platforms, rule system to drive the dynamic adaptation, etc).

1.2.1   Raising the level of abstraction: Towards Continuous Design

In the 6th century BC, Sun Tzu emphasized the role of reflection in the *Art of War* [154]:

> *If you know your enemies and know yourself, you can win a hundred battles without*
> *a single loss.*
> *If you only know yourself, but not your opponent, you may win or may lose.*
> *If you know neither yourself nor your enemy, you will always endanger yourself.*

This citation is obviously not about software engineering. However, reflection [23, 94] is a well-admitted concept to support dynamic adaptation. This citation can be transposed to the modern field of software engineering for adaptive systems. Here, "yourself" would be the *adaptive system itself*, and the "opponent" or "enemy" would be the *execution context* in which the system evolves. Finally, the "battles" would be the *dynamic reconfigurations* of the system. With an accurate knowledge about the context and the system itself, it is possible to take good decisions and make the system to safely migrate from its current configuration to a target configuration, to better cope with the changing context.

More recently, Daniel G. Bobrow and colleagues [23] defined reflection as:

> *The ability of a program to manipulate as data something representing the state of*
> *the program during its own execution. There are two aspects of such manipulation:*
> *introspection and intercession.* **Introspection** *is the ability of a program to observe*
> *and therefore* **reason about its own state***.* **Intercession** *is the ability of a program*
> **to modify its own execution state** *or alter its own interpretation or meaning.*

It consists in maintaining a causal connection between reality (*i.e.* the running system[2]) and a model of the reality[3]. Any relevant change in the running system will update the

---

[2]"Base-level" in the reflective terminology.
[3]"Meta-level" in the reflective terminology.

model, and any modification of the model will affect the running system. This contradicts Jeff Rothenberg's definition of what is a model since this reflective model does not avoid the danger and irreversibility, because of the strong synchronization between reality and model.

We acknowledge reflection is a fundamental concept to engineer adaptive systems, but we propose to go one step further in this thesis in order to include Jeff Rothenberg's arguments. The model should always reflect what happens in the running system (introspection), similarly to a mirror, so that it is possible to always reason on an up-to-date model. However, we should be free to manipulate, transform, validate, etc the model without directly modifying the running system. This is basically how human intelligence (and artificial intelligence [135] to some extent) works. It mentally builds several potential models of the reality and mentally evaluates these models by means of *what-if* scenarios [18][4]: what would happen if I would do *this action*? During this mental reasoning, the manipulation of the model does not impact the reality, until an acceptable solution has been found. Then, this solution is actually realized, which has an impact on the real world. In other words, the mental model is re-synchronized with the reality. In the case where a relevant aspect of the reality changes during the reasoning process, the model is updated and the reasoning process should re-build mental models, ideally by updating already existing models.

Keeping design models at runtime makes it possible to abstract the reality, by hiding irrelevant details, but also makes it possible to keep some design information that are usually lost at runtime. This way, models can enrich the reality with meta-information that would ease the decision making process.

Another aspect of the reality that should be abstracted is the execution context in which the running system evolves. The gap to bridge between the runtime and the model space is rather large. Indeed, sensors integrated in the runtime generate (quasi-) continuous flows of raw values, with very little abstraction. At design-time, designers often use qualitative values [33, 51], such as *high*, *medium* or *low*, to specify the environment of a DAS and its adaptation logic.

1.2.2 Variability Management, Product Derivation

As we already mentioned, fully specifying and implementing the state machine driving the execution of an adaptive system rapidly becomes a daunting and error-prone task because of the combinatorial explosion of the number of configurations, and the quadratic explosion of the number of transitions[5].

While it is still possible to specify this state machine for a small embedded system [14, 165], it is almost impossible for complex adaptive system, with millions or even billions

---

[4]See the sidebar by Bran Selic.

[5]There exist N(N-1) potential transitions among N configurations, if we do not count self transitions, which should normally produce no dynamic adaptation.

of configurations [51, 105]. Even in the case where states and transitions are specified at a high-level of abstraction, it rapidly becomes difficult for a human, and even for a machine, to specify such a huge number of artifacts [51, 105].

Recently, Sven Hallsteinsen and colleagues conceptualized DAS as Dynamic Software Product Line [70, 69] (DSPL) in which variabilities are bound at runtime. Similar to traditional Software Product Lines [38] (SPLs), the idea is to capitalize on commonalities and properly manage variabilities among the different products. This way, the huge number of configurations a DAS should deal with is described in intention, rather than in extension.

In a SPL, products are mostly derived by human decisions, from requirement to deployment-time [66], to fit a particular need; however, the derivation process of a DSPL is more complex and more diverse. Unlike a "classic" SPL, the products of a DSPL are highly dependent: the system should dynamically and safely switch from its current configuration to another one. The decision of producing a new product (*i.e.* to adapt to another configuration) depends on the type of adaptive system: in Self-Adaptive Systems (SASs), *e.g.* adaptive systems embedded in planes [118], the derivation process is fully automated and often follows the autonomic MAPE (Monitor-Analyze-Plan-Execute) loop [81]. In user-driven adaptive systems (UDASs), *e.g.* a house-automation system [116], the choice of the dynamic feature to integrate is mostly done by the end-user.

The Software Product Line (SPL) community [11, 38, 169] already proposes formalisms and notations, such as feature diagrams, to describe a wide range of products in intention rather than in extension. The fundamental idea is to describe the commonalities between products and to offer constructs (alternatives, options, n-among-p choices, etc) and constraints (mutual exclusions, requires, etc) to properly describe the variabilities among products. This way, designers do not need to enumerate all the possible products. However, decomposing a system in terms of commonalities and variabilities requires efficient, consistent and expressive composition mechanisms to be able to actually compose a selection of variable features into the common features, to obtain the final product.

The Software Composition (SC) community (which encompass Aspect-Oriented Software Development (AOSD), Feature-Oriented Software Development (FOSD), Component-Based Software Development (CBSD), etc) proposes different but complementary composition mechanisms, such as aspect weaving, mixins, feature composition, component composition, etc to compose software artifacts. Recently, many approaches propose to implement [80, 5, 6, 48, 97] (code level) or design [88, 77, 64, 102, 100, 120] (model level) SPLs using composition techniques.

1.2.3   Making the Adaptation Loop Explicit

Separation of concerns [119] is a well-admitted best practice in software engineering. In adaptive systems, the adaptation logic should be separated from the business logic [12, 10, 14, 40, 52, 122, 123, 133, 134, 165, 168], to improve the comprehension, the maintain-

ability, the testability and the modularity of such systems. Beyond the separation of the adaptation and business logic, it is important to clearly separate the components realizing the business logic from the components realizing the dynamic adaptation. According to Betty Cheng and colleagues [35]:

> *Understanding and reasoning about the control loops of a self-adaptive system is integral to advancing the engineering of self-adaptive systems' maturation from an ad-hoc, trial-and-error endeavor to a disciplined approach.*

Considering control loops as first class entities, makes it possible to reason about the architecture of control loops, their properties (stability, reactivity, performances, etc), and configure (and even dynamically reconfigure) control loops according to the needs.

1.2.4   On the Importance of Handling Several Reasoning Paradigms

Most approaches for engineering adaptive systems only propose one paradigm to express the adaptation logic. The most common reasoning paradigm is ECA (Event-Condition-Action) [40, 41]. It basically consists in specifying for well defined context fragments, which actions to execute. For example, if a fire is detected, sprinklers must be activated. As noticed in [51], each individual rule is quite easy to specify, understand and implement: it can be seen as a set of *if-then* constructs. However, fully specifying an adaptive system using ECA rules often requires to define numerous rules [51]. Moreover, one specific context can trigger several rules (since conditions are defined on context fragments), which can be conflicting. It thus requires to specify additional constraints (priorities, exclusions, etc) to properly manage the set of ECA rules defining the adaptation logic of an adaptive system.

Another common way to express the adaptation logic is to define goals that the system should (tend to) reach [51, 62, 34]. For each feature of the system, the designer should precise how it impacts QoS properties. He should also specify which properties to optimize in which contexts. At runtime, the system should find the best selection of features, which best optimize the properties that are important in the current context. As noticed in [51] specifying a system using goals is rather intuitive and simple. Indeed, the choice of the features to integrate is left to the system itself. However, multi-dimensional reasoning algorithms are often resource and/or time-consuming.

By analogy, the human body has two main "reasoning" capabilities: reflex[6] and thinking[7].

ECA rules can be seen as a kind of reflex, since they do not really involve any reasoning. In the case where the system is in a critical context, they can quickly reconfigure the

---

[6]From Wikipedia: *A reflex is an involuntary and nearly instantaneous movement in response to a stimulus, mediated via the reflex arc, which do not pass directly into the brain, but synapse in the spinal cord.*

[7]From Wikipedia: *Thinking allows beings to model the world and to deal with it according to their objectives, plans, ends and desires.*

system into an acceptable configuration. Goal-based decision techniques perfectly match the definition of "thinking". Depending on the resource and the time allowed to reasoning, the system can evaluate different configurations and find the one which offers the best trade-off.

ECA-rules and goal-based rules have complementary benefits and drawbacks. On the one hand, ECA rules can efficiently be processed at runtime. However, it rapidly becomes difficult to fully specify an adaptive system using ECA rules. On the other hand, goal-based rules allows specifying the adaptation logic at a higher level of abstraction. However, processing these rules at runtime is often more costly. Ideally, it should be possible to combine several reasoning algorithms in order to leverage their respective advantages, while limiting their respective drawbacks.

1.2.5   Reactivity *Versus* Stability

One important and obvious criteria of an adaptive system is its ability to adapt according to external stimulus (execution context, user preferences, etc). However, dynamic adaptation is not immediate and inevitably disturb the running system, since some components should sometimes be stopped and restarted to be able to safely migrate from one configuration to another.

The context in which an adaptive system evolves is very dynamic and can potentially change more rapidly than the adaptive system. QoS properties such as the bandwidth can be very fluctuating and depends on many parameters, most of them being out of control: number of systems that use the network, size and quantity of data exchanged, quality of the network, etc. In the worst case where the bandwidth, or any other context variable, fluctuates around a threshold which triggers a reconfiguration, this could make the adaptive system to continuously oscillates between two configurations. Even worst, if the fluctuations of the bandwidth are quicker than the adaptation process, this could make the adaptive system always in a configuration that is not adapted to the current context. In this case, this would be preferable to switch to a safe mode *w.r.t* the bandwidth, and let the other variation points open. Here, the safe mode would be to consider that the bandwidth is low, even if it is sometimes high, for short periods.

Specifying and implementing an adequate trade-off between reactivity and stability is not simple. Improving the stability can be achieved by abstracting the context, using for examples time windows, hysteresis cycles [155], complex event processing [93], fuzzy logic [87] etc. Such an abstraction can hide context fluctuations, thus offering a more stable basis for reasoning. However, it could also hide critical contexts where the system must adapt. To achieve an acceptable reactivity, it is often necessary to operate at a lower level of abstraction, with little analysis, to be able to quickly react according to the context.

1.3   Contributions

In this thesis, we propose to leverage models and Model-Driven Engineering [139] (MDE) techniques, not only at design-time but also at runtime [18], in order to consider and abstract relevant aspects of a DAS: its (dynamic) variability, its context, its adaptation logic and its architecture. By operating at this high level of abstraction, we argue that it is possible to reason efficiently (on a reduced space), by hiding irrelevant details, and automate the reconfiguration process using MDE techniques.

In this thesis, we will particularly focus and contribute on the following questions:

1. How to actually build the configuration (architecture) corresponding to a set of features, with no need to fully specify all the possible configuration beforehand, while preserving a high degree of validation?

2. How to plan and executes the changes from the current configuration to a newly produced configuration, with no need to write low-level platform-specific reconfiguration scripts by hand?

1.3.1   Modeling Adaptive Systems

In the context of the DiVA European project, we have modeled four dimensions/aspects of an adaptive system [50]:

- its variability: describing the various features of the system, and their natures (options, alternatives, etc)

- its environment/context: describing the relevant aspects of the context we want to monitor (environment), as well as the current context.

- its adaptation logic: describing when the system should adapt. It consists in defining which features (from the variability model) to select, depending on the current context using adapted formalisms (rules, goals, etc).

- its architecture: describing the configuration of the running system in terms of architectural concepts.

1.3.2   Aspect-Oriented Modeling to Support Product Derivation

The SPL community [11, 38, 169] already proposes well established formalisms and notations, such as feature diagrams, to manage variability by describing a range of product without enumerating all the products. The SC community proposes a wide range of different but complementary techniques, which can be leveraged to derive products from a product line model [80, 120, 106]. We propose to rely on this background to design the variability of DASs, or DSPLs, and derive configurations from a variability model.

Since we intensively rely on models, we naturally propose to use Aspect-Oriented Modeling (AOM) techniques in order to refine and compose features [120, 64, 160], as we proposed in [106, 105, 104]. AOM relies on a strong theoretical background, such as the graph theory [24], which make it possible to perform early validation at design-time, without enumerating all the possible configurations. It is for example possible to detect dependencies or interactions [78] between features.

While the literature is quite prolific about AOM approaches [9, 88, 102, 78, 64], few implementations are actually publicly available. In this thesis, we present how we leverage the SmartAdapters approach [88, 101] we developed in the INRIA Triskell group. However, our approach is not tied up to a particular model weaver, making it possible to use different weavers with different composition capabilities, or to seamlessly make the approach evolve to integrate new weavers.

1.3.3   Decoupling and Synchronizing Reflection Model from/with the Reality

We previously made an analogy between adaptive systems and human intelligence, and its ability to build (when needed) mental models reflecting the reality, not directly coupled with the reality. This thesis will not deeply explore this analogy. However, we argue this is an important capability modern dynamically adaptive systems should integrate. Modern DAS are too complex so that it is difficult to generate, validate, simulate, etc all the possible configurations of a DAS at design-time. But it is also unconceivable in many domains (health-care, critical systems, etc) to reconfigure a system into a configuration that has not been previously validated, when human lives directly depend on software systems for example.

We will detail how MDE techniques allow building configurations which do not affect the reality, and how to automatically synchronize suitable models with the running system, with no need to write low-level platform-specific reconfiguration scripts. If a configuration is invalid, it is simply discarded, with no need to perform a roll-back on the running system.

The ability of a DAS to build models not directly connected with the reality makes it possible to validate all the configurations where the system will transit, **when needed**, with no need to specify and pre-validate all the possible configurations of a DAS *a priori*. This online validation provides a high-degree of confidence in the adaptive system: the adaptation process will never make the system transit through a configuration that has not been validated. Online validation is a first step towards high assurance liable complex adaptive systems, where all the configurations cannot be validated at design-time. Note that this thesis does not aim at contributing on the validation process itself. Instead, by decoupling and synchronizing the reflection model from/with the reality, it makes it possible to apply existing techniques and integrate future advances in the validation domain.

1.3.4 A Reference Architecture to Support Model-Driven Dynamic Adaptation and Continuous Design

The ideas presented in this thesis are actually implemented and integrated into a reference architecture, which supports Dynamic Adaptation and Continuous Design (to some extent). The main tasks driving the initial configuration and subsequent reconfigurations of an adaptive system are encapsulated as components. These components mainly exchange models, which describe the different dimensions/aspects of the running system itself as well as of its execution context. We leverage these models to fully automate the deployment and reconfiguration of adaptive systems [105, 106, 104]. At any time, it is possible to modify the models specifying the DAS.

This reference architecture has been successfully employed and demonstrated in several contexts:

- Several live demonstrations have been performed at the Models@Run.Time workshop [114, 111, 103] and at the AOSD conference [107].

- It has been integrated as the configuration and reconfiguration engine of the home-automation middleware developed by the Triskell team, mainly by Grégory Nain.

- It is currently used by the industrial partners of the DiVA European project[8] to implement their case studies: An airport crisis management system (Thales[9]), and a next-generation customer relationship management system (CAS Software AG[10]).

- Since the reference architecture is itself a component-based system, it is easy to bootstrap *i.e.*, to use an instance of the reference architecture to deploy and dynamically adapt another instance of the reference architecture. In other words, it makes it possible to dynamically adapt the adaptation process. This idea will be discussed as one of the main perspectives of this thesis.

1.4 Organization of this Thesis

This thesis is organized as follows. Chapter 2 presents a State-of-the-Art of approaches for managing the dynamic variability of adaptive systems, with a particular focus on Model-Driven and Aspect-Oriented approaches. Part II presents the different contributions of this thesis and validate these contributions. Chapter 3 presents the different models we leverage to design an adaptive system. Chapter 4 explains how we use Aspect-Oriented Modeling techniques to refine and compose the dynamic features of the system. Chapter 5 details how we bridge the gap between the runtime and the model space by abstracting

---

[8]http://www.ict-diva.eu
[9]http://www.thalesgroup.com
[10]http://www.cas.de/english

the execution context and realizing a causal connection that totally prevents architects from writing low-level reconfiguration scripts. Chapter 6 shows how the different contributions are integrated into a reference architecture and Chapter 7 validates our contributions on two case studies. Part III concludes this thesis and presents some perspectives.

Chapter 2

State of the Art on Dynamic Variability Management

Contents

tel-00538548, version 1 - 22 Nov 2010

This chapter presents representative approaches for dynamic variability management. We first present a brief overview of some adaptive execution platforms, which support the last step of the dynamic variability management process: the dynamic adaptation itself. After introducing the main ideas behind Model-Driven Engineering (MDE), we then present relevant model-based approaches for developing adaptive systems, which provide a higher level of abstraction than the adaptive platforms. Finally, we present approaches that propose a clear separation of concerns between the adaptation logic and the business logic, which allow designers to encapsulate dynamic reconfigurations into self-contained units. We also present approaches that port AspectJ-like aspects to component-based platforms to encapsulate cross-cutting concerns not easily captured by components.

2.1   An Overview of Some Adaptive Execution Platforms

This section presents three representative execution platforms, for executing component-based applications (Fractal), or service-oriented applications (OSGi) or applications that leverage both paradigms (SCA).

2.1.1   Fractal

Fractal [29, 21, 2, 90] is a modular and extensible component model part of the OW2 consortium, to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces. The most famous implementations of Fractal are Julia and AOKell (Java), Cecilia (C), FractNet (.NET) and FracTalk (SmallTalk). Other component models in that category encompass for example OpenCOM [19, 20].

The Fractal Component Model

The Fractal component model supports the definition of primitive and composite components, bindings between the interfaces provided or required by these components, and hierarchic composition (including sharing). Primitive components contain the actual code (standard classes), and composite components are only used as a mechanism to deal with a group of components as a whole, while potentially hiding some of the features of the subcomponents. Components of the same level can be linked by bindings on compatible interfaces. A composite component can export the provided or required interfaces of some of its internal component.

Fractal Controllers

Each Fractal component consists of two parts: a controller which exposes the interfaces of the component, and a content. In the case of a primitive component the content is a class. In the case of a composite component the content is an assembly of components. All interactions between components pass through their controller. The model thus provides two mechanisms to define the architecture of an application: bindings between interfaces of components, and encapsulation of a group of components into a composite. Fractal supports two kinds of dynamic adaptation: parameterization and reconfiguration, which are realized by 6 controllers that may be present in components:

- **Attribute controller** and **Name controller**: it allows getting and setting the internal attributes of a component and allows getting and setting the name of a component.

- **Binding controller**: it allows accessing to bindings (owned by client components) and bind/unbind client interfaces to/from server interfaces.

- **Content controller**: it allows accessing the content of a composite and add/remove sub-components into/from the composite.

- **Lifecycle controller**: it allows accessing the state of a component and starting or stopping it.

- **Super controller**: it allows accessing the super components containing the component. Note that Fractal allows shared components and a components can thus have several super components.

Additional controllers can easily be defined in Fractal. In Julia, controllers are defined using Mixins whereas AOKell propose to use AspectJ or Spoon to implement them.

A Transactional Framework for Fractal Reconfigurations

In [91, 92, 42], Léger *et al.* propose a transactional framework for reliable dynamic reconfigurations for Fractal. They propose to equip reconfiguration scripts with ACID[1] properties, inspired by the work performed a few decades ago in the database community [67]. The atomicity is achieved by a roll-back mechanism, which consists in undoing all the reconfiguration actions in the reverse order. Most reconfiguration actions indeed have a reverse action, for example *bind* and *unbind*. However, some actions cannot be so easily reversed. Typically, the *start* action executes some initialization treatment. Let us imagine a simple reconfiguration that deploys two components and bind them to already deployed component. In this simple scenario, the first component is correctly deployed and sends an SMS to a user, to notify him that new features are available. However, if the

---

[1]Atomicity, Consistency, Isolation and Durability

second component is not correctly deployed, a roll-back is performed: the first component is removed. In this simple example, it is not trivial to correctly undo the *start* action.

The consistency is realized by means of integrity constraints, defined at three levels using the FPath language [41]:

- **Model Level**: A generic set of constraints specific to the Fractal component model *e.g.*, a component must be unbound before it is removed. It is important to note that such constraints are defined as pre-condition on the operations defined in the Fractal intercession API. In other words, these constraints are checked at runtime, while executing the reconfiguration script (before executing each reconfiguration action).

- **Profile Level**: A generic set of constraints, which restrict the Fractal component model. For example, a profile can forbid the use of shared components.

- **Application Level**: A set of constraints specific to the a given application. For example, all the composite component must contain a logger component.

The isolation is realized using a standard lock mechanism, and the durability is achieved by the use of a journal, which keeps track of all the transactions, both in memory and on disk.

2.1.2   OSGi

The OSGi[2] (Open Services Gateway initiative) consortium is composed of famous companies from different domains: automotive industry (BMW, Volvo), mobile industry (Nokia), e-Health (Siemens), SmartHome (Philips), etc. It provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle.

Typically, an OSGi bundle (component) is responsible for managing its dependencies by itself. It can try to set its dependencies when it is starting, by searching required services from the OSGi service registry, or by registering to the OSGi event mechanism to be notified when required services appear or disappear. For example, the following code fragment sets the *helloworld* reference when the client component starts.

```
1  /*
2   * Client Component, in OSGi
3   */
4  public class Client implements BundleActivator, IClient {
5
6    private BundleContext context;
7    private IHelloWorld helloworld;
8
```

---

[2]http://www.osgi.org

```
9   public void start(BundleContext context) {
10    this.context = context;
11
12    //registering the component as IClient
13    context.registerService(IClient.class.getName(), this, null);
14
15    //setting the helloworld reference
16    ServiceReference[] refs = context.getAllServiceReferences(null,
17      "(objectClass="+IHelloWorld.class.getName()+")");
18    helloworld = (IHelloWorld)context.getService(refs[0]);
19  }
20  }
```

OSGi provides very flexible and powerful mechanisms for managing the lifecycle of components. However, since the dependencies should be handled inside the components themselves, it is very difficult to separate the business logic from the adaptive logic and implement complex adaptation policies involving several collaborating components.

2.1.3 SCA

The Service Component Architecture is a standard proposed by OSOA[3] (Open Service Oriented Architecture) to develop and deploy distributed service-oriented applications. This technology agnostic standard leverages concepts from the SOA and the CBSE communities. It is supported by famous companies such as IBM, Oracle or SAP. Well-known implementations of this standard are Apache Tuscany[4], Newton[5] (built upon OSGi) and FraSCAti[6] (built upon Fractal).

SCA supports many programming languages for component and service implementation, and provides various communication modes (asynchronous, MOM, RPC). It also provides different types of binding to easily interact with legacy components or services: Web Services, EJB, JMS, JCA, RMI, RPC, CORBA, etc. The added value of SCA is to clearly separate the implementation of services from the wiring logic of a service based application.

The SCA standard focuses on the development and deployment steps. It provides almost no specification for the dynamic adaptation of applications at runtime, nor for the dynamic adaptation of the platform itself. This important lack (in the context of adaptive systems) it filled by the FraSCAti implementation [140]. Another important point addressed by FraSCAti is the architecture of the platform itself, which is not specified in the standard.

FraSCAti relies on the Fractal component model (hence its name) to introduce introspection and reconfiguration capabilities that the application or the platform itself can benefit.

---

[3]http://www.osoa.org/display/Main/Service+Component+Architecture+Home
[4]http://tuscany.apache.org
[5]http://newton.codecauldron.org
[6]https://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti

2.1.4   Discussion

Modern execution platforms like Fractal [29, 21, 2] OSGi or SCA provide clear API for introspection and reconfiguration of software systems, which are the two main concepts of reflection [94, 23]. However, there is no clear distinction between the reflection model and the reality. Modifying the model implies modifying the reality: there is no mean to preview the effect of a reconfiguration before actually executing it, or to execute *what-if* scenarios to evaluate different possible configurations, etc. This lack of an explicit and independent reflection model makes it difficult to perform validation *a priori* (before actual reconfiguration). Verification is thus performed at runtime (*e.g.* pre-condition on reconfiguration actions, as proposed by Léger [91, 42]) during the reconfiguration process itself, which rolls back if it encounters a problem. We claim that an explicit and independent model, which can be re-synchronized later on, would make it possible to perform most of these verifications before the actual reconfiguration, thus preventing the system to try to adapt to an erroneous configuration, and finally rollback to its initial state. However, the ability to roll-back a reconfiguration (being actually executed) is still needed, because of unanticipated problems, such as hardware failures. These conclusions are summarized in Table 2.1.

Support of Reflection	Explicit Model	Level of Abstraction	Validation a priori	Validation during Reconf.
Yes	No	Imperative style using the reconfiguration API *i.e.*, a hand-written reconfiguration script that contains all the atomic reconfiguration actions, that should be carefully ordered to ensure constraints (life-cycle exceptions, dangling bindings, etc).	No	Possible

Table 2.1: Summary of features of adaptive execution platforms

2.2   Model-Based Development of Dynamically Adaptive Systems

2.2.1   A brief overview of Model-Driven Engineering

The fundamental idea of MDE [139] is to consider models as first-class entities, which abstract some aspects of the reality for a given purpose (see Rothenberg's definition in the Introduction chapter). Following the MOF pyramid-shaped structure illustrated in

Figure 2.1, each model conforms to a well-defined metamodel that describes the concepts and relationships of a given domain. Each metamodel conforms to MOF, which is self-described (bootstrapped). A metamodel is the central part of a Domain-Specific Modeling Language (DSML). It defines the abstract syntax of the DSML.

M3 Level (bootstrapped)
MOF, E-MOF, ECore

M2 Level (Metamodels)
UML, any domain metamodel

M1 Level (Models)
Class diagrams,
state machines, etc

conforms to

conforms to

conforms to

conforms to
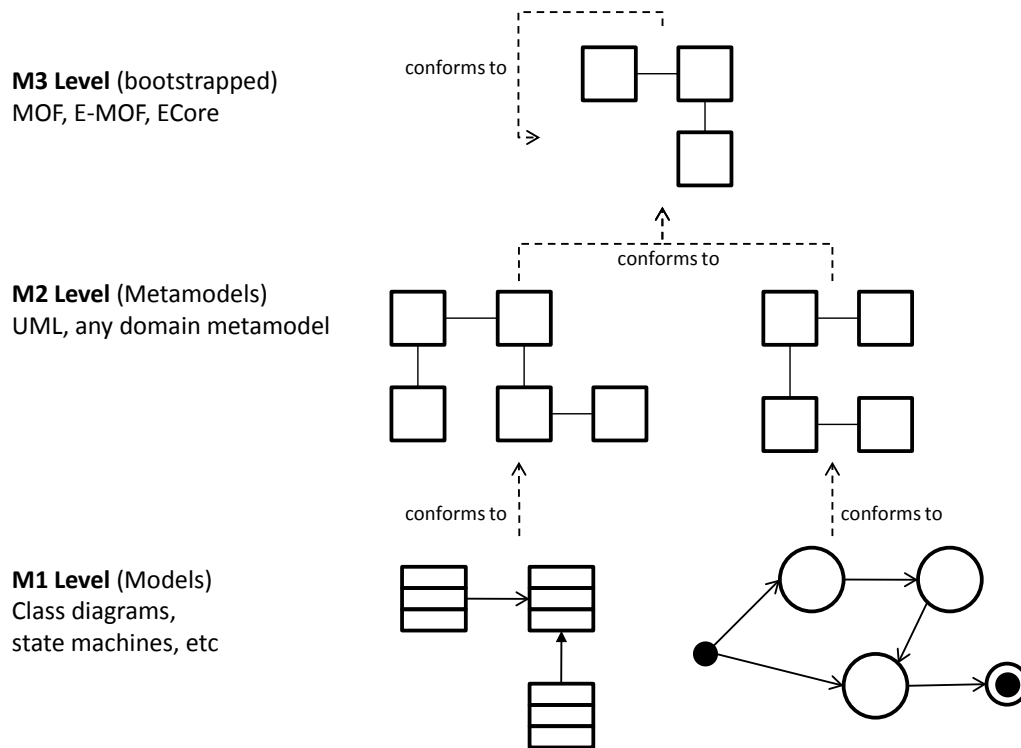
Figure 2.1: The MOF hierarchy

The de-facto standard to design metamodels is the Eclipse Modeling Framework (EMF)[7]. EMF generates an API associated with the metamodel to programmatically (in Java) build and modify models. It also generates the code of a default reflexive editor to visualize and edit models using a tree-editor. It is possible to use tools like EMFText[8] to describe the textual syntax of a DSML, or the Graphical Modeling Framework (GMF)[9] to describe the graphical syntax.

In addition to the syntax of a DSML, two important aspects of a DSML are its static and its dynamic semantic. Kermeta [112][10] provides a Model-Oriented and Aspect-Oriented

---

[7]http://www.eclipse.org/modeling/emf/

[8]http://www.emftext.org/

[9]http://www.eclipse.org/modeling/gmf/

[10]http://www.kermeta.org/

meta-programming environment to specify the static and the dynamic semantic of models using an OCL-like syntax. The static semantic consists in defining constraints that are not directly captured in the metamodel that models must ensure. Similarly to OCL, Kermeta allows defining invariants and pre/post-conditions. The dynamic semantic consists in defining the behavior of operations defined in a metamodel. This for example allows simulating models at early stages of the development life-cycle.

Another key point of MDE is the ability to manipulate and transform models using:

- Model to Model transformations (M2M): A M2M transformation is a program that transforms an input model into an output model. The input and output models can have the same metamodel, or different metamodels. There exist several model transformation languages, such as Kermeta (imperative style), or QVT or ATL (declarative style).

- Model to Text transformations (M2T): A M2T transformation is a program that transforms an input model into text (usually, source code). It is possible to use any model transformation language to generate text. Tools dedicated to M2T include for example MOFScript or KET (Kermeta Emitter Template).

A more detailed background on MDE is presented in the background chapter of Steel's PhD Thesis [144].

2.2.2 Extensive Model-Based Development of DAS

In [165, 167], Zhang and Cheng present a model-based approach for the development of dynamically adaptive software systems. They mainly focus on the behavior of such systems by clearly separating the design of adaptive and non-adaptive behavior, using state-based modeling languages like Petri nets. They introduce the notion of Simple Adaptive (SA) program/model, illustrated in Figure 2.2, and define it as a triplet (S, M, T) where S is the source program/model (before adaptation), M is a set of adaptations and T is the target program/model (after adaptation). Consequently, they define a general adaptive program/model as the union of all the SAs needed to describe the whole system.

Figure 2.2: A Simple Adaptive System, as defined by Zhang and Cheng [165]
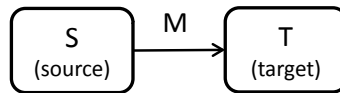
The conditions (*e.g.*, a predefined change in the environment, a user action, etc) under which a system should adapt are specified in the transitions. In [165], the authors give

an example of a pointcut-like condition that triggers an adaptation, in the context of a GSM application: "*after encoding a packet and before sending the packet, and after the data in the compressed data buffer have been shifted to the next location*".

Adaptations can be triggered when the system (its state-based model) has reached a quiescent state *i.e.*, a state *s* in **S** such that there exists a state *t* in **T**, $s = f(t)$, and any execution path including the adaptive transition *s* to *t* does not violate any global invariant. More intuitively, adaptations can be triggered if the system is in a stable state and the path from the initial configuration **S** to the target configuration **T** does not cause any trouble. The runtime adaption mechanism depends on the code generation. If the source code is a strict refinement of the model, then the executing system will behave like its state-based model. Zhang and Cheng distinguish three type of adaptations [166] that are formalized using temporal logic [44]:

- **One-point adaptation**: after receiving an adaptation request, the source system adapts to the target mode when it reaches a safe state.

- **Guided adaptation**: after receiving an adaptation request, the source system restrains its source program and triggers the adaptation when it reaches a safe state.

- **Overlap adaptation**: the target behavior starts when the source behavior is still active in a restrained environment aiming at ensuring the safety of the transitions.

Bencomo *et al.* propose the Genie approach [15, 12, 13, 14, 17] for managing the dynamic variability of DAS. It leverages two types of diagrams, illustrated in Figure 2.3:

- A Transition Diagram, specifying the whole state machine defining the adaptive system, similarly to Zhang's approach. A State is a possible configuration of the adaptive system, and a transition is a possible migration path between two configurations, which is associated with a first order logic predicate on the context. This state machine explicitly reifies the Event-Condition-Action (ECA) rule system driving the evolution of the DAS. This corresponds to the notion of general adaptive system defined by Zhang and Cheng [165].

- A Variability Diagram, specifying the functional variability of the DAS. This is a simplified feature diagram describing the different variation points of the system. For each variation point, the designer specifies a list of variants (which realize the variation point) as well as an operator controlling these variants: alternative, n-among-p choices, etc. Unlike the transition diagram, the variability diagram describes the whole system in intention, rather than in extension. Each variant is linked to all the states of the transition diagram corresponding to configurations that integrate this variant.

Both the approaches of Zhang *et al.* [165] and Bencomo *et al.* [15, 12, 13] require the explicit enumeration of all possible configurations and relevant transitions between these
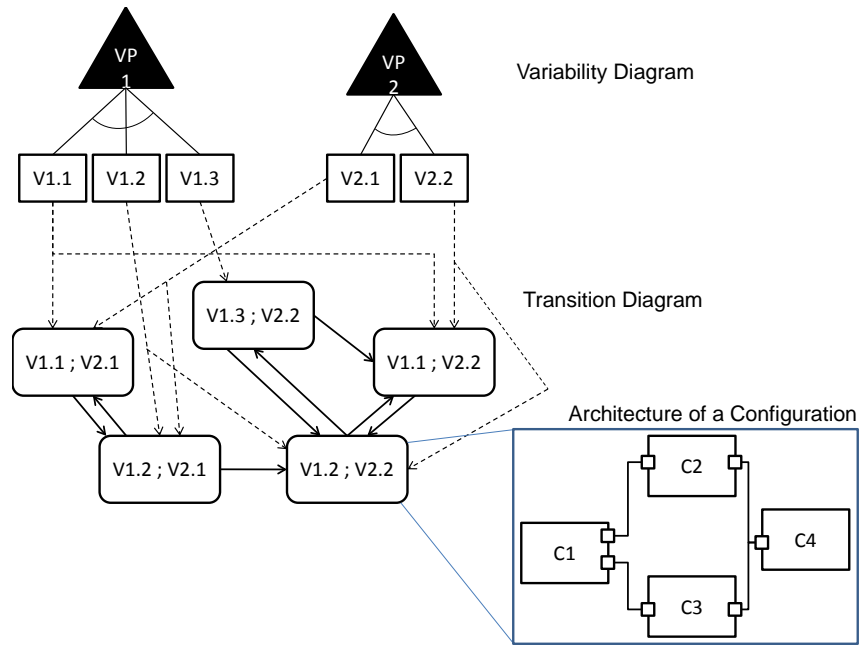
Figure 2.3: Variability and Transition Diagram in the Genie approach

configurations. This has the main advantages of making automation possible (such as code generation) and provides the means to reason about behavioral correctness and consistency of the DAS, before, during, and after adaptation [168, 95]. However, the scalability is the main drawback of such approaches. The number of configurations to specify grows in a combinatorial way with the number of variable features of the system. In the worst case, card(configurations) = $2^{card(features)}$. Moreover, the number of transitions among these configurations evolves in a quadratic way: for N configurations, it may exists up to N(N-1) transitions.

2.2.3   Some multi-staged approaches to dynamic adaptation

In [71], Heaven *et al.*present an approach for goal-driven architectural adaptation. Inspired by the robotics community [58], they propose a 3-layered approach to dynamic reconfiguration:

- **Goal management**: This layer handles high-level goals and sophisticated reasoning mechanisms to synthesize tasks from these high-level goals. This layer is realized by the mean of an extension to the Labeled Transition System Analyzer (LTSA) [95].

- **Change management**: This layer deals with adaptations within the current task. It interprets the plans of the goal management layer to actually execute these plans by selecting a suitable configuration of components.

- **Control**: This layer is where the components are executed. Components are implemented in Java, following the Backbone component model [96].

The system embedded in an environment is modeled by a domain model, which is a state machine describing all the sequences of actions that the system has to execute according to the context[11]. This state machine is generated by LTSA from a set of actions, a set of fluent propositions (describing properties of the environment, and the actions that trigger these properties to *true* or *false*), and a set of constraints (expressed in LTL, which impose constraints on the order of system actions depending on the context properties). This state machine is pruned according to the goals of the system, which are defined as a predicate on the states and the transitions of the state machine *e.g.*, all the states where a given fluent is true and all the transitions that allows reaching these states with the shortest path (from the initial state). From this pruned state machine, it is possible to determine several paths that reach the goals *i.e.*, several sequences of actions.

From a set of actions, the component configuration is built in three steps:

1. **Dependency Analysis**: this step consists in choosing the components that can realize the needed actions and to connect these components on relevant and compatible ports [149]. This step constructs a set of candidate configurations.

2. **Structural Constraints**: this steps checks that the candidate configurations all the constraints provided by the user. This typically consists in checking architectural style *e.g.*, ComponentA $\in$ *arch* $\Rightarrow$ ComponentB $\in$ *arch*. These constraints cannot always be ensured by construction by the dependency analysis. Configurations that do not ensure the constraints are completed, if possible, so that they finally ensure the constraints.

3. **Utility Check**: all the valid configurations are finally evaluated with respect to their impact on QoS properties (power drain, reliability , etc). Each component can be annotated with property (*e.g.* power-drain = 500 mA) and the user has to define a utility function for each property along with a weight. A utility function is a function $u : property\ range \rightarrow [0..1]$, which defines the utility of a component according to its associated property. For example, the utility (*w.r.t.* power) of a component with no power drain is 1, whereas the utility of a component with a power-drain of 1A is 5%. The utility of a component is thus the weighted sum of all its "atomic" utilities. Finally, the utility of the configuration is defined as the average utility among all its components.

---

[11]This state machine is thus different from the state machine describing the dynamic adaptation of the adaptive system

The most useful configuration is finally leveraged to drive the dynamic reconfiguration process and adapt the running system to make it switch to this new configuration.

In [99, 98] Ben Mokhtar *et al.*propose an approach to QoS-based service composition. Each client describes a service orchestration with a state machine automaton, where an edge corresponds to the invocation of an operation provided by a service provider, and is associated with a set of QoS property. Each QoS property (availability, latency and cost) is associated with an evaluation rules, which depend on the different constructs we can find in an automaton: sequence, choice, simple loop and dual loop. For example, the availability of a sequence of operations *op1*, *op2* is **availability(*op1*) * availability(*op2*)**; the latency of the same sequence is **latency(*op1*) + latency(*op2*)**, etc. These evaluation rules of very similar to utility functions. The dynamic adaptations correspond to a dynamic selection of service providers, which is realized in three steps:

1. a list of *matching services* is build, using a subsumption relationship between required and provided services

2. a *service selection* identifies the most suitable services among the list of matching services. This step allows filtering servers that provide too much services, not required by the client, with potential dependencies with other services not requested by the client

3. the different configurations are computed by composing the provided services (and associated QoS) with the client orchestration. This gives an estimated QoS for each configuration, which allows selecting the best one.

2.2.4   Designing DAS as Dynamic Software Product Lines

The work by Bencomo *et al.*was a first step towards designing DAS as DSPLs [69, 70] since they are using notations and formalisms from the SPL community. However, they still require the designer to describe all the possible configurations of the DAS in extension.

In [30, 31, 32], Cetina *et al.* propose an approach to autonomic computing through the reuse of variability models at runtime. They propose to use feature diagrams to describe the functional variability of a smart-home *e.g.,* volumetric detector, visual alarm, lightning by occupancy, etc. Each feature is refined in terms of components and communication channel using the *superimposition* operator. For each feature, the superimposition operator returns the architectural elements realizing this feature. This operator is a simple explicit *1-to-n* mapping where the source is a feature and the targets are all the architectural elements realizing the feature. This requires all the architectural elements to be defined beforehand. In practice, Cetina *et al.* need to define all the architectural elements in one single model, corresponding to the union of all the possible configurations of a DAS. Indeed, since an architectural element can contribute to different features it is difficult

to clearly separate the feature refinements (expressed in terms of architectural models) into distinct models, using the superimposition operator. The limited expressiveness of the superimposition operator makes it difficult to express features that depend on others and/or that crosscut others in a modular way. For example, a simple log feature is typically composed of a central log component and all the other components, which require logging facilities must be connected to this log component. The manually specified superimposition operator associated to the log feature should return the log component as well as all the bindings. In the case where a new feature, which requires the log, is introduced after a refactoring of the feature diagram, this also requires the designer to modify the log feature to connect the newly introduced components to the log component.

The selection of features is based on the ECA paradigm. Each feature is directly tagged with context conditions, which determine when the feature must be active or not. They define the notion of architecture increments and decrements using the superimposition operator to determine the architectural elements that should be added or removed in order to switch from a source configuration to a target configuration. These increments and decrements are then transformed into a reconfiguration plan, which is executed in order to actually reconfigure the running system.

2.2.5 Synchronizing Runtime and Models using MDE

Very recently, several approaches from the "models@runtime" community [114] propose ways to synchronize high-level models with the runtime. Some approaches mainly focus on monitoring (runtime → model), while some other approaches focus on the Model-Driven dynamic adaptation of software systems, with a bi-directional connection. These approaches do not propose to manage the dynamic variability as such, but propose ways to reason at a higher level of abstractions and/or to automate the dynamic adaptation process once a configuration has been derived.

In [156], Vogel *et al.* propose an approach for incremental model synchronization for efficient runtime monitoring. The running system is observed by a set of sensors, which allows building a first model of the running system, with little abstraction. Using model transformations, based on Triple Graph Grammars (TGG) [61], they propose to map this low-level source model to different target models, which are more abstract and more focused. Target models for example include the architecture, the performance or the failures. TGG uses 3 graph grammars to achieve the transformation from the source model to a given target model:

1. a graph grammar describing the source model

2. a graph grammar describing the target model

3. a graph grammar describing the mapping from the source model to the target model

The source model is updated, either in a push or pull model, depending on the API provided by the runtime sensors. Each model transformation listens the modifications of the source model, using the EMF notification mechanism, to determine if and how the target model should be updated. This way, both the source model and the various target models are not re-built from scratch each time a change affects the runtime system. However, Vogel *et al.* do not provide ways to synchronize some of the target models with the source model (in the other direction), nor to synchronize the source model with the running system.

In [143], Song *et al.* present an approach for generating bi-directional synchronization engines between running systems and their model-based views. The developers have to specify which elements have to be managed (monitored and/or dynamically adapted), and how to manipulate these elements using the API provided by the execution platform. Using these two inputs, they automatically generate a synchronization engine able to dynamically maintain a model representing the running system. This makes it possible to use MDE tools usually employed at design-time, to manage applications at runtime. To automate the code generation of the synchronization engine, they require the execution platform to provide a clear monitoring and management API, which is the case for most modern platforms like J2EE, Fractal, Android, etc, and a one-to-one mapping between runtime concepts and model elements. This means that each model is platform-specific *i.e.*, there exist one metamodel for each execution platform. While this makes the causal connection more straightforward, and probably more efficient, it significantly hinders the reuse of tools for checking, simulating, visualizing, etc across different execution platforms. In other words, specific checkers, simulators, editors, etc should be created for each specific platform.

Waignier *et al.* present an approach for bridging the gap between design (architecture) and runtime [126, 158, 157, 159], with a particular emphasis on the use of models@runtime for debugging and validation purposes. They use different metamodels to describe the structure and behavior of component-based applications, as well as contracts on the structure and the behavior that the applications must ensure. Using the structural metamodel, they can describe the architecture of a component-based application and leverage this model to automatically deploy and reconfigure an application. This is achieved by a model comparison, which determines the components, connectors and validation points (for debugging/validation) that are added or removed.

Fractal systems can be managed by the Jade framework [27, 25] where management operations are effected both on a (Fractal) model of a running system (model@runtime), and on the running system itself (a Fractal system, native or wrapped). It thus makes it possible to reason on and manipulate a "fake" version of the running system before executing a reconfiguration script. Jade thus makes it possible to preview the impact of

a reconfiguration by actually executing it the "fake" system. However, Jade does not provide high level languages for the designers to actually implement the reconfiguration. Designers still need to directly use the Fractal API or a low level language like FScript.

In [60], Georgas *et al.* propose to use architectural models to manage and visualize dynamic adaptation. Similarly to other approaches presented earlier in this chapter, they use ECA rules to define the adaptation logic of a DAS. Using their Architectural Runtime Configuration Management (ARCM) approach, they propose to trace and visualize the dynamic adaptation of a software system, at runtime. Similarly to the work of Bencomo *et al.*, ARCM uses a state machine to represent how the system dynamically adapt. However, unlike the work of Bencomo *et al.*, this state machine is not specified at design-time. Instead, this state machine is automatically built and maintained at runtime. ARCM first associates an initial state to the configuration initially deployed. When a dynamic reconfiguration occurs, ARCM create a new state corresponding to the new configuration and an edge linking the target and source states. If a configuration (resp. a reconfiguration) has already been constructed (resp. triggered) the corresponding state (resp. edge) is reused, to avoid the duplication of similar states and edges in the state machine. The states and the edges are associated with meta-data such as counters or time-stamps. The initial and current states are high-lighted, so that it is easy to see in which configuration the system was started and what is its current configuration. The edges holds the reconfiguration script that makes it possible to actually switch from the source configuration to the target configuration. It consists in collecting all the actions of all the ECA rules that were triggered, since a context change can potentially trigger several ECA rules.

To overcome the limitation of OSGi, Spring DM (Dynamic Modules)[12] and BluePrint propose to manage the configuration of OSGi applications using a declarative XML-based mechanism to dynamically add, remove, and update modules in a running system. Moreover, it has the ability to deploy multiple versions of a module simultaneously. Typically, a Spring bean (component) is a POJO (Plain Old Java Object) with getters and setters for the reference that can be accessed and set. Unlike OSGi, components are not responsible for setting their dependencies by themselves. On the contrary, these references are set from the outside by SpringDM, by calling the appropriate getters/setters.

2.2.6 Discussion

There is a growing interest around the development of dynamically adaptive systems. Several communities have recently joined efforts to bring new ideas in order to tame the complexity of DAS. In particular, the Model-Driven Engineering and the Variability Management communities have put significant efforts in applying their techniques to the development of DAS. This has lead to the creation of two workshops, associated with the

---

[12]http://www.springsource.org/osgi

reference conferences of these 2 domains: *Models@Run.Time* workshop [114] (at MOD-ELS, since 2007) and *Dynamic Software Product Line* workshop (DSPL) [68] (at SPLC, since 2007). These 2 workshops are complementary to the *Software Engineering of Adaptive and Self-Managing Systems* (SEAMS) workshop [45] (at ICSE, since 2005-06).

Some Model-Driven approaches for the development of DAS require the enumeration of all the possible configurations and transitions of a DAS. MDE allows reducing the complexity by providing a high degree of abstraction, validation and automation *e.g.* code generation or automatic synchronization of the models with the reality. However, in the case of complex adaptive systems, the huge number of configurations and transitions to specify rapidly causes scalability issues. It soon becomes necessary to describe this dynamic variability in intention rather than in extension, in order to reduce the number of artifacts to specify. This is the idea of DSPL, which uses well known notations and formalisms, such as feature models, to describe the variability of DAS. To actually realize the dynamic variability of DAS, the variability model describing the DAS must be derived, in order to obtain one configuration. Cetina *et al.* [30, 31, 32] for example propose to use a rather simple composition operator to derive the variability model.

Different from the previous approaches, the "multi-staged" approaches of Heaven *et al.* [71] and Ben Mokhtar *et al.* [99, 98] mainly focus on service orchestration rather than directly focusing on dynamic reconfiguration. From a given orchestration (directly specified by a client or inferred from high-level goals by a predicate on a state machine) they propose to determine the corresponding architecture, or selection of services, in several steps. In [71], the global state machine the possible sequences of actions is generated from other artifacts. In [99, 98], a "template" state machine describing the service orchestration needed by each client has to be specified, which is later on substituted with actual provided services. While the number of artifacts to specify at design-time does not explode, the techniques used at runtime explore the whole space of configurations to find the best one, which can rapidly causes scalability issues.

Table 2.2 summarizes these conclusions. We claim that emerging "models@runtime" approaches offer a good basis for using more advanced composition techniques during the derivation process. Moreover, with an independent reflection model, it should be possible to ensure a level of validation close to the extensive model-based approaches, with no need to specify all the configurations *a priori*.

Approach	Variability Management	Validation	Trigger
Extensive Model-Based	Explosion of the number of artifacts to specify	Extensive validation at design-time	ECA, "hard-wired" in the model
Multi-staged	Good management of the artifacts to specify. At runtime, exploration of the configuration space to find the most adapted one.	During the different steps of the process	A change in high-level goals (a predicate on the state machine) and utility functions.
Cetina *et al.*	Good management, hindered by a rather limited *superimposition* operator	At design-time, on the feature model	ECA
Models ↔ Runtime	Explosion of the artifacts to specify (if not used in combination with an approach dedicated to variability management)	Possible to apply usual MDE techniques	None (if not used in combination with an approach dedicated to self-adaptation).

Table 2.2: Summary of features of model-based approaches

2.3 Separation of Concerns to clearly separate the Adaptive Logic From the Business Logic

In [136], Sadjadi *et al.* propose the notion of transparent shaping to migrate legacy system into DAS. Transparent shaping makes heavy use of reflection, aspect-oriented programming and also involves code generation at run time to achieve this migration.

The remainder of this section presents approaches which leverage reflection and encapsulate reconfigurations into separate concerns, and approaches which propose to dynamically weave AspectJ-like aspects.

2.3.1 Encapsulating Reconfigurations as Separate Concerns

David *et al.* propose SAFRAN [40], an open-source extension of the Fractal component model to support the development of self-adaptive components, *i.e.* autonomous software components which adapt themselves to the evolutions of their execution context.

SAFRAN is composed of three sub-systems on top of Fractal:

- **FScript** [41] is a domain-specific language used to program the component reconfigurations which will adapt the application. It provides a custom notation which

makes it possible to navigate in a Fractal architecture and offers certain guarantees on the changes applied to the target application, for example the atomicity of the reconfigurations.

- **WildCAT** [39, 26] is a generic toolkit to build context-aware applications. It is used by SAFRAN policies to detect the changes in the application's execution context, which should trigger adaptations. WildCAT (v2) is based on the Esper [1] event monitoring framework that allows designers to query, aggregate and sort events using a SQL-like syntax. Note that SAFRAN is based on a previous version of Wild-CAT, which does not integrate Esper.

- **a Fractal controller** that binds FScript and WildCAT through the reactive rules of adaptation policies. These rules follow the Event-Condition-Action pattern, where the events are detected by WildCAT and the actions are FScript reconfigurations. The adaptation controller allows the dynamic attachment of SAFRAN policies to individual Fractal components and is responsible for their execution.

SAFRAN proposes to clearly separate the adaptation logic from the application logic. The adaptive behavior is seen as a particular aspect of the system. Such a separation of concern eases the evolution and maintenance of the adaptive behavior. Using an aspect-oriented approach, the possible configurations of the system are not explicitly described. The whole possible set of configurations is defined in intention by ECA rules. However, as noticed in [51], fully specifying an adaptive system by means of ECA rules can rapidly becomes difficult, because of the interactions between rules. SAFRAN proposes no mechanism to detect or resolve conflicts and interactions between rules.

Similarly to Fractal, SAFRAN does not propose to maintain a higher-level representation of the running system. Consequently, it is difficult to determine the impact of a reconfiguration before actually performing this reconfiguration. Even if FPath and FScript propose to abstract the API provided by Fractal, developers still have to write verbose reconfiguration scripts. These scripts manipulate low level primitives that have to be ordered in order to produce consistent scripts.

In [57], Garlan *et al.* present Rainbow, an architecture-based framework to support self-adaptation of software systems. It allows designers to define adaptation policies that are triggered when the associated invariant is not respected. Similarly to SAFRAN or Plastik, adaptation rules are based on the ECA paradigm. In a later work [37], they admit that ECA alone is not able to properly deal with multiple goals that a DAS needs to cope with, because the number of cases to consider grows intractable and because of the difficulty to define and maintain meaningful trade-offs between potentially conflicting rules. To better reason about the adaptation logic, they make a clear distinction between *observable* and *actionable* states of a DAS. The observable state space is the infinite set of states we can observe using probes integrated in the runtime. Typically, a bandwidth sensor will

provide the instant value of the bandwidth every seconds, in kbits/s. Even if this value is bounded by the physical medium of the network, there exist (in theory) an infinite number of real numbers in this interval. The actionable state space is a finite set of states, relevant for decision making. Typically, the bandwidth is discretized as {high, medium or low}. This drastically reduces the set of states to consider.

To reduce the effort needed to implement trade-offs between conflicting goals, they propose to extend their ECA rule system with goals. They also define goals as enumeration *e.g.*, response time = {low, medium or high}. Each goal is associated with a relative importance (a real in [0..1]). Finally, each ECA rule additionally describes its impact on the objectives. Depending on the context, the most useful ECA rule is triggered. This simply consists in computing a weighted sum for each rule.

The MADAM European project[13] and its follow-up MUSIC[14] focus on providing techniques and tools to reduce the time and effort to develop self-adaptive mobile applications [132, 133]. They rely on the notion of component framework to describe their applications [53]. A component framework is an assembly of component types *i.e.* a template of architecture where component types will be substituted by actual implementations. Quite similarly to Rainbow, the choice of the actual implementations of the component types is realized via goal policies, expressed as utility functions. Each component implementation is associated with some predictors, which precisely specify the impact of a particular implementation on (QoS) properties. For example, the response time of a given component implementation could be defined as follows: *response =* **if** *context.bandwidth > 80* **then** *10* **else** *10 + 100 * (80 - context.bandwidth)/80)*. Finally, a global utility function (which can aggregate intermediate utility functions) computes the overall utility of the application. This way, the system can evaluate different configurations and choose the most useful one, using brute force (*i.e.*, by exploring the space of possible configurations). In [134], the MUSIC approach is used to weave AspectJ-like aspects, similar to aspects in FAC (see next subsection).

In [59, 164] Georgantas *et al.*present their work carried out in the AMIGO European project[15]. They are particularly interested in uncontrolled distributed reconfiguration in pervasive computing systems, based on the SOA paradigm. They rely on WSAMI [76], a lightweight Web Services middleware suitable for mobile devices with limited resources, as the underlying platform. They define the notion of pervasive configuration to deal with the dynamic reconfigurations of pervasive applications, which is supported by various entities (mobile or stationary) available in the environment:

- a set PS of provided application services, defined in WSDL and BPEL

---

[13]http://www.ist-music.eu/MUSIC/madam-project
[14]http://www.ist-music.eu/MUSIC
[15]http://www.hitech-projects.com/euprojects/amigo/

- a set PR of orchestration processes, defined in BPEL

- a service discovery service SD, which periodically checks the environment for other instances of SD, and maintains a registry

- a process execution engine PEE, which constitutes the reconfiguration manager (RM) that is completely distributed across the pervasive configuration,

- a changes detection service D, which is a simple push-based notification service,

- a checkpointing service CH, which requires the web-service to describe their conversations in a set of atomic sub-conversations that can easily be roll-backed.

- a recovery service RE responsible for managing the roll-back mechanism, and possibly

- a state transfer service ST to ensure the consistency of stateful services before and after an adaptation.

These entities realize a distributed reasoning in the context of service-oriented pervasive computing systems that drive the dynamic adaptation of such application.

2.3.2   AspectJ-like Weaving in Component-Based Systems

A Brief Overview of AspectJ

The idea of AspectJ is to propose a new modularization unit (called aspect) to encapsulate cross-cutting concerns that cannot properly be captured in the OO paradigm. The goal of the aspects is to reduce the tangled and scattered code due to cross-cutting concerns, in order to ease the comprehension and maintenance of applications.

AspectJ is a popular Aspect-Oriented Programming language, which extends Java with AO concepts:

- **Join Point**: point of interest where an aspect can be woven. For example: a call to a method, get/set on an attribute.

- Pointcut: defines a set of join points. For example: all the call to the method *m* of the class *C*. It describes (possibly with quantification) **where** the aspect will be woven.

- Advice: extra-behavior that extends (or replaces) the former behavior, at all the places (join points) that match the pointcut. It describes **what** the aspect brings and **how** it modifies the base behavior.

An aspect is thus a pair <Pointcut, Advice> that will impact all the join points intercepted by the the pointcut, at compile-time or load-time.

Some Approaches Mixing AOP and CBSE

Pessemier *et al.* propose FAC (Fractal Aspect Component) [121, 122, 123, 124, 125], an open-source aspect-oriented extension to the Fractal Component Model. It combines Component-Based Software Development (CBSD) and Aspect-Oriented Programming (AOP) by integrating CBSD notions into AOP, and vice-versa.

FAC introduces new aspect-oriented structures into the Fractal platform: Aspect Component (AC), Aspect Domain (AD) and Aspect Binding (AB). An Aspect Component is a regular component that encapsulates a cross-cutting concern providing advice pieces of code as services. Advice interface is a server interface to which an aspect can bind. A weaving interface provides a control interface for the setting and ordering of aspect bindings. Aspect components are Fractal components supporting the weaving interface. An Aspect Binding is a binding that links an AC to other components. Finally, an AC and all the aspectized components bound via ABs constitute an Aspect Domain. Note that FAC leverages the notion of shared components provided by Fractal to allow component to be contained into several ADs.

An aspect component, like a regular component can provide or require services via interfaces. Additionally, an aspect component can declare advice interfaces for declaring cross-cutting concerns.

An aspect domain is associated to each aspect component and encapsulates all the components aspectized by a given aspect component. In fact, aspect domains are Fractal composite components. Note that components can be shared by several composite components, and consequently a component can be aspectized by several aspect components.

FAC aspect components are woven at all the join points that match the associated pointcut i.e., weaving or not weaving an aspect only depends on the topology of the running system. There are no other conditions, such as environment or QoS, which can trigger the weaving of the aspect. In that sense, FAC proposes mechanisms to encapsulate and weave cross-cutting adaptations at runtime, but no language to express adaptations policies.

Pessemier *et al.* define a pointcut language for matching method calls, as follows: **JP Type** ; **Component** ; **Interface** ; **Method**, where:

- **JP Type** (join point type) indicates the type of the calls we want to match i.e., outgoing (CLIENT keyword), incoming (SERVER keyword) or both (no keyword).

- **Component**, **Interface** and **Method** are regular expressions for referring to components via their names, interfaces via their names and methods via their signatures.

For example, the pointcut <CLIENT C* ; * ; add*> refers to all outgoing calls of any method whose name starts with add, encapsulated in any interface of any component whose name starts with C. Then, all the components that match this pointcut will automatically be aspectized by the Aspect Component associated to this pointcut, using an Aspect Binding. An AC and all its aspectized components form an Aspect Domain.

In FAC, an advice is a piece of code with the following structure: <before statements> <proceed> <after statements>, where:

- proceed is a call to the former service, before interception

- before/after statements represent the additional code that is introduced before/after the former code. All these statements are optional.

If proceed does not appear in the code, it means that the former service is totally replaced. So, it is possible to modify the behavior of services by adding pre/post-treatment, or replacing the service by a new one. This definition of pointcut and advice is inspired by AspectJ [84, 83].

Note that the pointcut language does not allow to describe pointcuts like: "find all the components A, B, such as A and B are connected on a given interface I, and component A requires interface J." In other words, each join point is limited to one given method of one given interface of one given component. It is not possible to describe additional constraints (e.g., A should be connected to B).

The advice interfaces of an aspect components implemented in FAC should implement the Advice interface that extends the Interceptor interface. The Interceptor interface is defined in the AOP Alliance API (http://aopalliance.sourceforge.net/), an open source initiative to define a common API for AOP frameworks.

Surajbali *et al.* propose AOpenCOM [148], which offers very similar mechanisms for the OpenCOM platform.

2.3.3   Using *Aspects of Assembly* to Dynamically Compose Applications

In the context of ubiquitous computing, Riveill *et al.* propose to use the notion of Aspect of Assembly (AoA) to dynamically compose applications [46, 47, 153, 152], based on the SLCA (Service Lightweight Component Architecture) [73] component model. The dynamic adaptation is mainly driven by the appearance and disappearance of devices in the environment of the user or by events (such as changes in the user preferences).

Similarly to FAC [122] or to AspectJ [84, 83], an aspect of assembly is composed of:

- **A pointcut** describing where (which components) the advice will be woven. More precisely, it is an error-prone string-based filter on ports of the assembly, describing predicate over ports and components where the components of the advice will be attached. The pointcut language used in the Aspects of Assembly uses AWK [3], a programming language dedicated to the processing of text-based data. Using this pointcut language, it is possible to select ports belonging to different components, based on their IDs and names (strings). However, similarly to FAC, it does not seem possible to express more complex patterns on an architecture. Indeed, during the join point identification step, each component is actually considered as a separate

Pointcut	join point 1	JP2	JP3	JP4
PCC1	BC1	BC3	BC6	BC15
PCC2	BC12	BC7	∅	∅
PCC3	BC17	BC9	BC13	∅

Table 2.3: An example of Join points in AoA. Columns containing an empty sign (JP3 and JP4) are not join points.

> pointcut. The (object) matching process is thus individually applied for each component, leading to a table like 2.3, where a line indicate all the base components (BC*) that match a pointcut component (PCC*). Using this approach, it is impossible that the choice of a join point for a given pointcut component can restrict the choice of a joint point for another pointcut component. In other words, it makes it impossible to describe constraints among different components of the pointcut. However, this simple object matching can be efficiently implemented, with no need to rely on unification techniques [146], which require back-tracking facilities.

- **An advice** describing an error-prone string-based list of components[16] and actions for connecting these components to the ones defined in the pointcut. For each identified join points (JP1 and JP2 in 2.3), the advice is duplicated according to a default rule that cannot be overridden. It is for example not possible to specify that a given component should be global (reused for all the join points), while some others should be duplicated per join points. Finally, all the instances of advice are merged into the base assembly.

The context-awareness needed in ubiquitous systems is achieved by an horizontal approach, similarly to SAFRAN or Plastik. Rather than using a global control loop, which monitors the whole system and takes global decisions for adapting or not the system, they equip each AoA with monitoring and decision capabilities, thus achieving a decentralized adaptation. When several AoA are selected, they are all merged into the base assembly in order to obtain the whole assembly adapted to the current context. This merging allows the detection and resolving of conflicts before actually adapting the running system.

2.3.4 Discussion

There are two main trends when using Aspect-Oriented techniques to dynamically reconfigure component-based systems. Some approaches like FAC or AOpenCOM propose to apply AspectJ-like aspects to components. It consists in intercepting the invocation of operations on the provided/required ports of components, and wrapping the associated

---

[16]Components are defined as Strings in the AoA metamodel, and not as a first class concept relating to the notion of component in the SLCA metamodel

code with *before* and/or *after* treatments. These approaches offer mechanisms to dynamically deploy and undeploy AspectJ-like aspects, but do not propose languages to specify when (which context) to deploy these aspects, which is important in the case of self-adaptive systems. However, since FAC and AOpenCOM consider aspects as components, it should be possible to reuse existing approaches for the reconfiguration of self-adaptive systems.

Some other approaches like SAFRAN, Plastik, *Aspect of Assembly* or Rainbow propose to consider reconfigurations as separate concerns, clearly separated from the business logic. These approaches propose different formalisms to determine when a given reconfiguration should occur. Most approaches use ECA rules, which basically consist in defining thresholds on context variables. Some other approaches use more elaborated goal-based rules, which let the system reason about the context to determine the best possible configuration. We acknowledge ECA and goals as two fundamental paradigms for describing and implementing the adaptation logic of DAS. Rather than opposing these two paradigms, we claim that the control loop [28, 81] governing a DAS should be able to seamlessly combine different reasoning paradigms.

Different from the approaches focusing on the reconfiguration of component-based applications (*e.g.* the work by Bencomo *et al.*, Garlan *et al.*, MADAM/MUSIC), the approach by Georgantas *et al.* addresses a different kind of adaptation: it mainly consists in adapting the service-based application depending on the apparition/disappearing of services in the environment.

Whatever the reasoning paradigm, the adaptation is often realized by means of low-level hand-written reconfiguration scripts. *Aspect of Assembly* is a first step toward a higher-level way of describing reconfiguration. However, in practice, it also require to define a list of components and actions (bind/unbind) between them. We claim that the latest advances in the domain of models@runtime (see 2.2.5) make it possible to use advanced model composition languages, usually employed at design-time. This way, it would prevent architects from writing error-prone reconfiguration scripts.

These conclusions are summarized in Table 2.4.

2.4   Aspect-Oriented Modeling to Support Model-Driven Software Product Lines

At the code level, Aspect-Oriented Programming (AOP) [84, 83, 7], and related programming paradigms such as Feature-Oriented Programming (FOP) [127] or Mixins [142], provide flexible ways to implement Software Product Lines [80, 5, 6, 48, 97]. The idea is to encapsulate each feature into a well-defined composition unit [151] (aspect, feature, mixin layer) that can easily extend an existing program. All these paradigms proposes different but complementary composition mechanisms in order to extend/refine Object-Oriented

programs.

Recently, Aspect-Oriented Modeling (AOM) has been applied in the context of Model-Driven Software Product Lines [88, 77, 64, 102, 100, 120]. In the context of the DiVA project, we pioneered the use of AOM in order to manage the dynamic variability of complex adaptive systems [105, 106, 104], or Dynamic Software Product Lines. AOM approaches tend to unify the composition concepts of AOP, FOP, Mixins, etc of the programming level in order to propose advanced composition mechanisms at design-time, based on model composition and model transformation. We can distinguish two main types of AOM approaches:

- Merge-based approaches, such as Kompose [49, 55, 56] or the Composition Directives [130], which propose systematic ways to merge models. Elements with the name signature, previously defined by the designer (in the case a default name matching is not sufficient), are merged into a single one, while elements that do not match are introduced into the resulting model. In the case where the two input models are not aligned, it is possible to apply directives (mainly renaming) to modify some elements and force the match. The *superimposition* operator used by Cetina *et al.* [30, 31, 32] can be seen as a simple merge operator. While the merge-based approaches are well adapted to weave non cross-cutting features, it seems very difficult (even impossible) to weave cross-cutting features that impact several places.

- Pointcut-based approaches, such as MATA [160, 77], SmartAdapters [88, 101] or GeKo [102], which allow to weave the same aspect in several places (join points) of the same base model. The aspect is totally defined according to the pointcut. At weaving time, a first pass will determine all the join points. Next, the aspect is contextualized for all the identified join points, and actually woven into the base model. Pointcut-based approaches are supposed to fill the lack of quantification of merge-based approach, so that they can weave cross-cutting aspects. However, it appears that these approaches do not offer the right mechanisms to properly weave aspects, as we further discuss in Chapter 4.

2.5   Conclusion

Figure 2.4 illustrates a map of the approaches we have presented in this state-of-the-art. These approaches are positioned according to their abilities with respect to validation and variability management. One of the ultimate goals of the development of adaptive system would be to achieve a degree of validation close to the extensive model-based approaches, with variability management capabilities close to the DSPL approaches and aspect-oriented approaches.
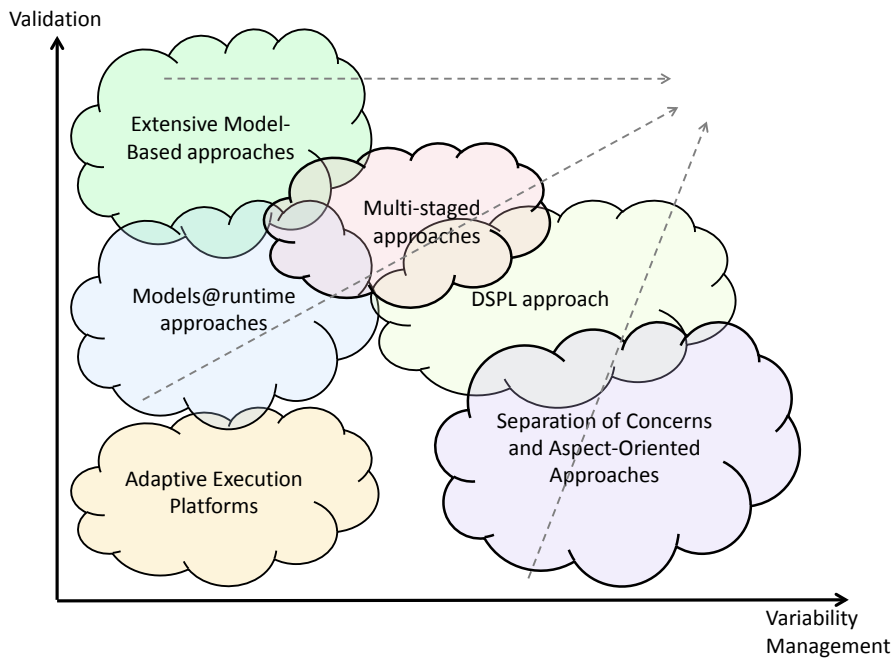
Figure 2.4:   A Map of the presented approaches

In this thesis, we combine Model-Driven Engineering and Aspect-Oriented Modeling techniques to go one step further towards this goal.

Approach	Variability Management	Validation	Trigger
SAFRAN, Rainbow, MADAM & MUSIC	Dynamic variability clearly encapsulated into adaptation rules. No need to explicitly specify all the possible configurations. The template architecture in MADAM/MUSIC rather restrict the possibility of adaptation.	No mechanisms to manage possible interactions between adaptation rules.	Either ECA or goals (utility functions).
Georgantas *et al.*	Distributed reasoning on the set of available services.	Recovery mechanism if the reconfiguration encouters a problem.	apparition-/disappearing of services in the environment.
AspectJ-like aspects for CBSE	Aspects clearly encapsulate (cross-cutting) variability units.	FAC and AOpenCOM are direclty based on Fractal and OpenCOM, with no explicit model allowing validation before adaptation.	Possible to use an existing reasoning paradigm to trigger aspect weaving/un-weaving.
Aspect of Assembly	Reconfiguration clearly encapsulated into *aspects of assembly*. Limited expressivity of the pointcut and advice languages.	An explicit architectural model is built before the actual adaptation, making it possible to detect and resolve errors.	ECA

Table 2.4: Summary of features of SoC approach

Part II

Contributions & Validations

Chapter 3

Models Manipulated and Exchanged

Contents

3.1 Introduction

This part details the contributions of this thesis, briefly introduced in the Introduction chapter. The key objective of the approach is to reduce the number of artifacts a designer has to specify to describe and execute an adaptive system, and to raise the level of abstraction of these artifacts. It is important to note that this thesis is not about self-adaptation mechanisms themselves. Instead, we propose to rely on the well established component composition as the underlying technique for dynamic adaptation.

We use a dynamic customer relationship management system (D-CRM) as a running example in this part. This system is inspired by one of the case studies of the DiVA European project, provided by CAS AG. The objective of the D-CRM is to provided accurate client-related information depending on the context. For example, when the user is working in his office, he can be notified by e-mail, via a rich web-based client. He can also access critical resources since he is connected to a trusted network. When he is driving his car to visit a client, he should be notified by a mobile or smart phone, and only with

information which is either critical or related to the client. If he is using a mobile phone, he can be notified via SMS or audio/voice, and phone calls are forwarded from his office. If he is using a smart-phone, he can additionally use a light-weight web client.

3.2 Overview

This section presents the different metamodels we leverage at design-time and at runtime to design and dynamically reconfigure adaptive systems (Figure 3.1) [50, 51].
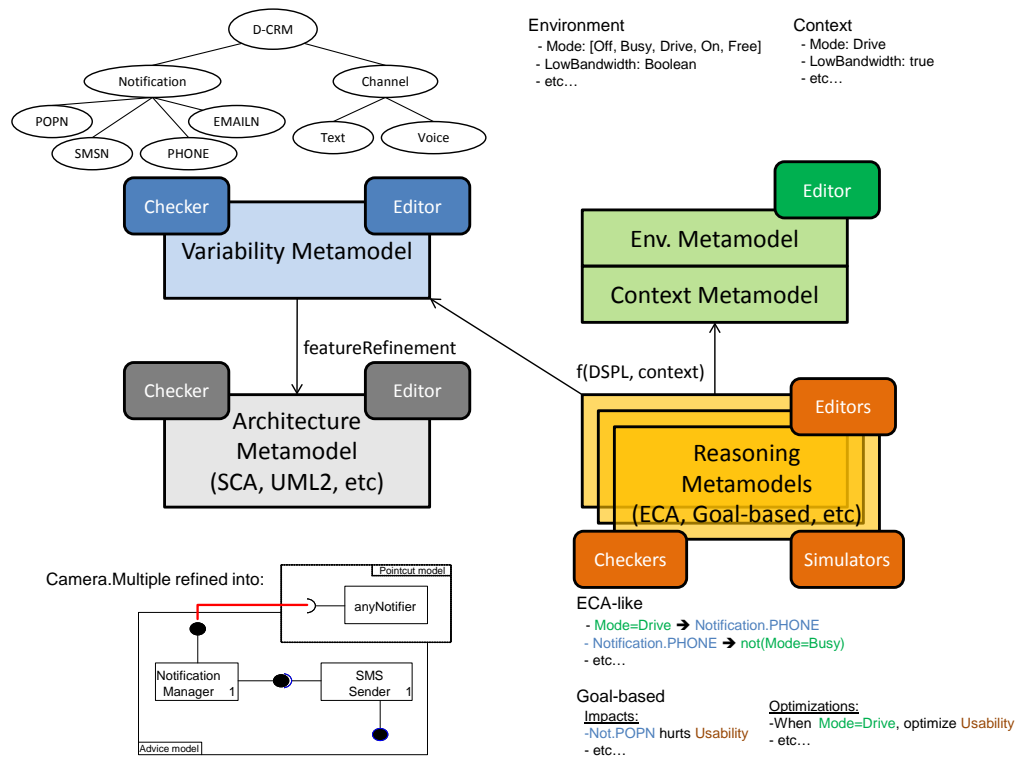

Figure 3.1: An overview of the Metamodels and Models

At design-time and runtime, we leverage 4 types of models described by 4 metamodels, which represent 4 fundamental aspects of a DAS:

- **Variability**: describes the different features of the system, and their natures (options, alternatives, etc)

- **Environment/Context**: describes the relevant aspects of the context we want to monitor (environment), as well as the current context.

- **Reasoning**: describes when the system should adapt. It consists in defining which features (from the variability model) to select, depending on the current context, using the appropriate formalisms.

- **Architecture**: describes the configuration of the running system in terms of architectural concepts.

The following sub-sections present a pragmatic solution for the different metamodels. These metamodels are designed with EMF. It makes it simple to develop graphical and/or textual editors (with GMF, EMFText, etc) and operate with other EMF-compliant tools, such as Kermeta [112] to specify the static semantic (OCL) and the dynamic semantic of these models, for validation and simulation purposes.

3.3 Variability Metamodel

The variability metamodel is illustrated in Figure 3.2. A *Dimension* corresponds to a variation point in the application and is associated with one or more alternative *variants*, which can realize this variation point. A *variant* is a fragment of functionality (feature) that can be included or not in the system. Each dimension declares a lower and a upper bound, specifying the number of variants that can be selected at a time.

This way, we can express the operators we usually found in feature diagrams:

- **option**: A dimension containing exactly one variant, with a [0..1] cardinality.

- **OR**: A dimension containing several variants, with a [1..n] cardinality. Optional OR can be obtained with a [0..n] cardinality.

- **XOR**: A dimension containing several variants, with a [1..1] cardinality. Optional XOR can be obtained with a [0..1] cardinality.

- **n-among-p**: A dimension containing p variants, with a [i..n] cardinality and $n \leq p$. In the case we want to choose exactly n features among p, i should be equal to n.

In addition to the cardinalities, which impose constraints within dimensions, we can also define constraints across dimensions. A variant can require or exclude another variant, defined in another dimension.

Any combination of variants, which respect the cardinalities and the constraints, is a valid configuration that can be used at a particular point in time.

The variability model of the D-CRM is illustrated in Figure 3.3. In this system, we are particularly interested in notifying the user from relevant information, using adequate mechanisms and channels. For example, the *Notification* dimension contains 4 alternative mechanisms: *Pop-Up*, *E-Mail*, *Text* and *Phone Call*. A valid configuration must exactly contain one ([1..1] cardinality) of these variants. Note that some variants define constraints.
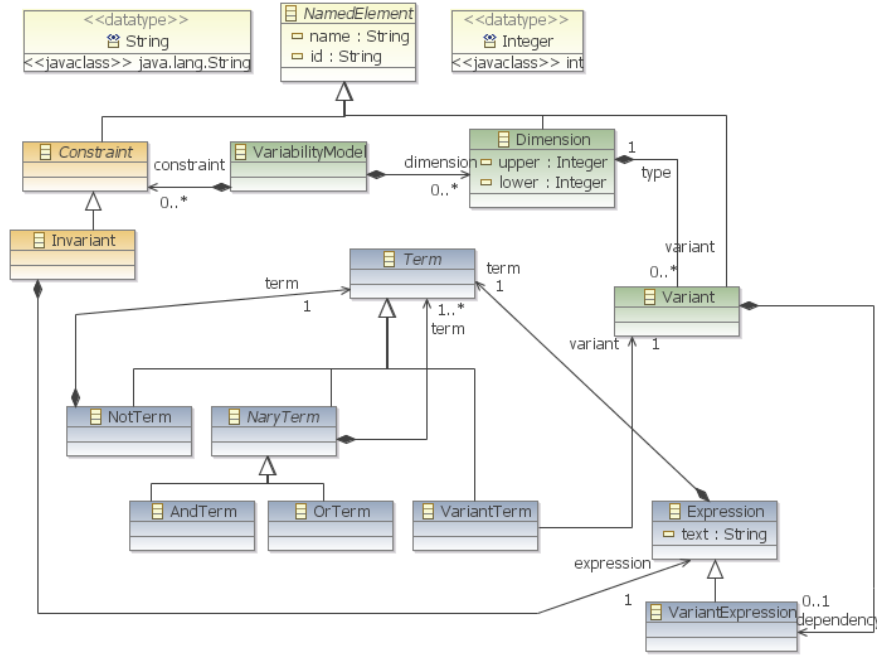
Figure 3.2: Variability Metamodel

For example, the *Text* variant requires the *Text CC* variant defined in the *Communication Channel* dimension. In other word, a configuration containing the textual notification mechanism *must* also contain the textual communication channel.

The number of possible configurations (without considering constraints and adaptation rules) can be computed as: $\prod D_i$, where D_i is a number associated to a dimension. Let D_i be a dimension with v variants and a $[n..p]$ cardinality, with $n \leq p \leq v$. Then, $D_i =$
$$\sum_{n \leq i \leq p} C_i^v$$
In particular:

- For an **option**, $D_i = C_0^1 + C_1^1 = 2$

- For an optional **OR** dimension $[0..v]$: $D_i = \sum_{0 \leq i \leq v} C_i^v = 2^n$. Indeed, each variant of the dimension can be seen as an individual option, which doubles the number of configurations.

- For a **OR** dimension $[1..v]$: $D_i = \sum_{1 \leq i \leq v} C_i^v = \sum_{0 \leq i \leq v} C_i^v - C_0^v = 2^n - 1$

	Name	ID	Lower	Upper	dependency
Dimension	Notification Mechanism	NOTI	1	1	-
Variant	Pop-up Window	POPN	-	-	RAI and TCC
Variant	E-Mail	MAILN	-	-	
Variant	Text	SMSN	-	-	TCC
Variant	Phone Call	PHON	-	-	VCC
Dimension	Comunication Channel	CC	0	-1	-
Variant	Text CC	TCC	-	-	
Variant	Voice CC	VCC	-	-	
Dimension	Ranking	RK	1	1	-
Variant	Default	DRANK	-	-	
Variant	Reduced	RRANK	-	-	
Dimension	Security	SEC	0	1	-
Variant	Weak security	WSEC	-	-	
Variant	Strong security	SSEC	-	-	
Dimension	Telephony	TEL	0	1	-
Variant	TAPI	TAPI	-	-	VCC
Variant	Skype	Skype	-	-	VCC
Variant	Jahjah	JAH	-	-	VCC
Variant	GSM	GSM	-	-	VCC
Dimension	User Interface	UI	1	1	-
Variant	Smart Phone	SPUI	-	-	TCC or VCC
Variant	Rich Ajax Interface	RAI	-	-	TCC
Variant	Voice Control	VUI	-	-	VCC
Dimension	Map Service	MAP	0	-1	-
Variant	Google maps	GMAP	-	-	TCC
Variant	Yellowmap	YMAP	-	-	TCC
Dimension	Video On Demand	VOD	0	1	-
Variant	Youtube	GVOD	-	-	(RAI or SPUI) and (TCC and VCC)
Variant	Dailymotion	DVOD	-	-	(RAI or SPUI) and (TCC and VCC)
Dimension	Cache	CACH	0	1	-
Variant	DefaultCache	DCACH	-	-	

Figure 3.3: Variability model of the D-CRM

- For an optional **XOR** dimension [0..1] with v variants: $D_i = C_0^v + C_1^v = 1 + v$

- For a **XOR** dimension [1..1] with v variants: $D_i = C_1^v = v$

In the D-CRM, this leads to 34,560 possible configurations if we do not consider the constraints.

3.4 Environment and Context Metamodel

The environment metamodel, illustrated in the top part of Figure 3.4, allows designers to specify the relevant aspects of the execution context that should be monitored. Each environment variable is either a boolean or an enumeration of pre-defined values, such as *high*, *medium* or *low*.

The context metamodel (bottom part) allows specifying values for the variables defined in the environment model. At runtime, the values of the variables are provided by context sensors and these may trigger a reconfiguration of the system. Note that a variable of the environment model is not necessarily associated with one sensor deployed at
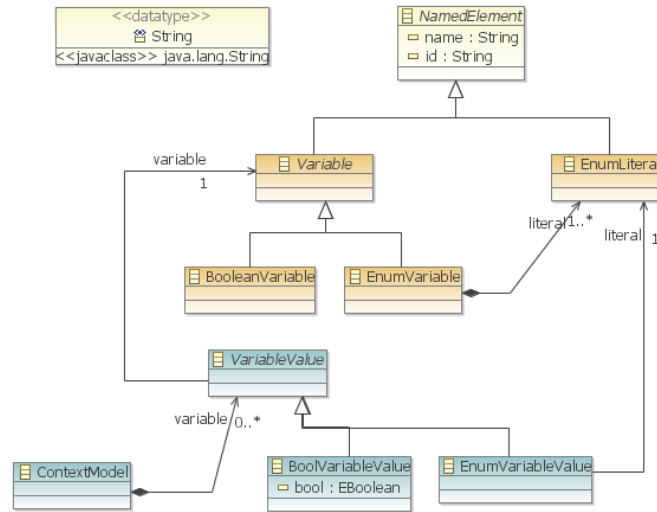
Figure 3.4: Environment and Context Metamodel

runtime. It rather gives a high-level view of the environment. The bridge between the actual sensors and the high-level variables can be filled by a monitoring framework like WildCAT (see Section 5.3).

The environment model of the D-CRM is illustrated in Figure 3.5. In the D-CRM, designers are interested in monitoring the power-level, the bandwidth, and need to be aware of the type of device (laptop, PDA, mobile phone) the user is actually using, as well as the presence of some external services (Google maps, Skype, etc).

		Name	ID	Values
⊞	Enum	Power level	POW	{LO, HI}
	Boolean	Low Bandwidth	LBW	-
⊞	Enum	Device	DEVTYPE	{MOB, PDA, LAP}
⊞	Enum	Calendar Mode	MODE	{OFF, BUSY, DRIV, ON, FREE}
	Boolean	Require map service	MAPR	-
	Boolean	Google maps available	GMAPA	-
	Boolean	Yellow maps available	YMAPA	-
	Boolean	Require vod service	VODR	-
	Boolean	Youtube available	GVODA	-
	Boolean	Dailymotion available	DVODA	-
	Boolean	Skype available	SKYPEA	-
	Boolean	Jahjah available	JAHJAHA	-
	Boolean	TAPI available	TAPIA	-
	Boolean	GSM available	GSMA	-

Figure 3.5: Environment Model of the D-CRM

3.5 Reasoning Metamodel

As we have seen in the State-of-the-Art chapter, there exist several formalisms such as Event-Condition-Action (ECA) rules [40], or Goal-Based Optimization rules [51]. An ECA rule system typically describes, for particular contexts, which features to select: **when** *context* **choose** *features*. A Goal-Based model typically describes how features impact QoS properties, using for example help and hurt relationships [62], and specify when QoS properties should be optimized (*e.g.*, when a property is too low).

We propose to define one reasoning metamodel for each paradigm. These metamodels typically reuse or extend the variability and the context/environment metamodels. We use two reasoning metamodels, as described in [51, 43].

3.5.1 ECA-like rules

ECA-like rules (Figure 3.6) link features with context fragments, expressed as logic predicate on context variables. We distinguish two types of relationships between features and context variables:

- **Require**: This type of constraints means that a given variant *must be* chosen in the specified context. Note that the variant *could also be* chosen in other contexts.

- **Available**: This type of constraints means that a given variant *may be* chosen (or not) only in the specified context. Note that the variant *cannot* be chosen in other contexts.
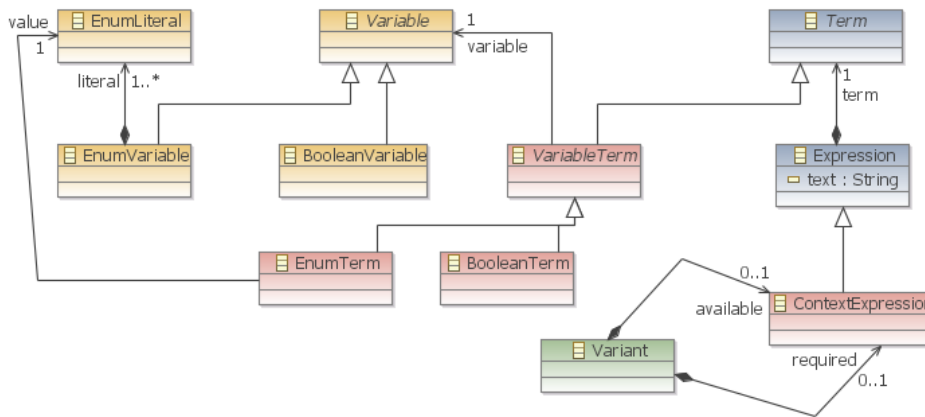
Figure 3.6: Metamodel for expressing ECA-like rules

More formally:

- a **require** rule is an implication: *context ⇒ variant*

- an **available** rule is an implication: *variant ⇒ context*

- the strict equivalence *context ⇔ variant* can be obtained by using a require rule in combination with an available rule.

Figure 3.7 illustrates the ECA-like rules of the D-CRM. For example, the *Phone Call* variant defined in the *Notification Mechanism* dimension is available in all the context except when the user is busy (*not(MODE = BUSY)*). Moreover, this variant is required when the user is driving. In other words, this variant *must* be present when the user is driving and *must not* be present when the user is busy.

	Name	ID	available	required
Dimension	Notification Mechanism	NOTI	-	-
Variant	Pop-up Window	POPN	DEVTYPE = LAP	
Variant	E-Mail	MAILN	DEVTYPE = PDA or DEVTYPE = LAP	MODE = BUSY and not DEVTYPE=MOB
Variant	Text	SMSN	DEVTYPE=MOB or DEVTYPE=PDA	(DEVTYPE = MOB and MODE = BUSY)
Variant	Phone Call	PHON	not (MODE = BUSY)	MODE=DRIV
Dimension	Comunication Channel	CC	-	-
Variant	Text CC	TCC	not MODE=DRIV	
Variant	Voice CC	VCC	not MODE=BUSY	
Dimension	Ranking	RK	-	-
Variant	Default	DRANK		
Variant	Reduced	RRANK	LBW or MODE=BUSY	
Dimension	Security	SEC	-	-
Variant	Weak security	WSEC		MODE=FREE
Variant	Strong security	SSEC	DEVTYPE=LAP and not MODE=ON	MODE=BUSY and DEVTYPE=LAP
Dimension	Telephony	TEL	-	-
Variant	TAPI	TAPI	(DEVTYPE = LAP and TAPIA)	
Variant	Skype	Skype	(DEVTYPE = LAP or DEVTYPE = PDA) and SKYPEA	
Variant	Jahjah	JAH	(DEVTYPE = LAP and JAHJAHA)	
Variant	GSM	GSM	(DEVTYPE = MOB or DEVTYPE = PDA) and GSMA	
Dimension	User Interface	UI	-	-
Variant	Smart Phone	SPUI	DEVTYPE = PDA	
Variant	Rich Ajax Interface	RAI	DEVTYPE = LAP	
Variant	Voice Control	VUI	(DEVTYPE = PDA or DEVTYPE = MOB)	DEVTYPE = MOB
Dimension	Map Service	MAP	-	-
Variant	Google maps	GMAP	(GMAPA and MAPR)	
Variant	Yellowmap	YMAP	(not (DEVTYPE = MOB) and YMAPA and MAPR)	
Dimension	Video On Demand	VOD	-	-
Variant	Youtube	GVOD	(VODR and GVODA)	
Variant	Dailymotion	DVOD	(VODR and DVODA)	
Dimension	Cache	CACH	-	-
Variant	DefaultCache	DCACH	MODE=BUSY or MODE=DRIV	MODE=BUSY or MODE=DRIV

Figure 3.7: ECA-like rules of the D-CRM

3.5.2 Goals

Goals (Figure 3.8) allows reasoning about the context and choose the most adapted variants, which optimize the goals.
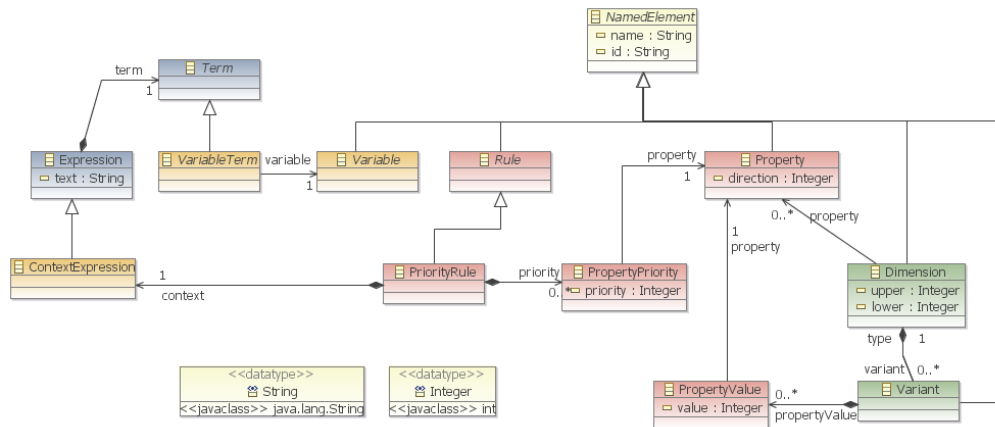
Figure 3.8: Metamodel for expressing Goal-based rules

To express goals, we define three pieces of information:

- **Properties**: The QoS properties relevant for the system, as well as a direction specifying if a given property must be maximized or minimized. Each dimension can specify if it has an impact or not on each property.

- **Impacts**: In the case where a dimension has an impact on a property, each variant can optionally specify how it impacts this property.

- **Optimization Rules**: These rules specify which properties should be optimized in which context. In the case several properties should be optimized, it is possible to specify a preference order. Unlike a classic ECA rule, it does not directly link the features to context fragment. Instead, it links high level goals (the QoS properties to optimize) to context fragment, letting the system to select the feature.

Figure 3.9 shows the impact of the variants on the QoS properties. We can for example see that the designer of the D-CRM estimates that the *Phone Call* variant has a medium cost, a low impact on the usability and is highly disturbing.

Figure 3.10 illustrates how the properties should be optimized depending on the context. For example, when the user is mobile, we want to optimize the power consumption (since the devices are probably running on batteries), and we also want to optimize (with a lower importance) the usability of the system.

	Power Consumption	Cost	Usability	Performances	Proximity Search	Security	Disturbance
⊟ Notification Mechanism (NOTI)	false	true	true	false	false	false	true
Pop-up Window (POPN)	-	-	High	-	-	-	Medium
E-Mail (MAILN)	-	-	Medium	-	-	-	Low
Text (SMSN)	-	Low	Low	-	-	-	Low
Phone Call (PHON)	-	Medium	Low	-	-	-	High
⊟ Comunication Channel (CC)	false	false	false	true	false	false	true
Text CC (TCC)	-	-	-	High	-	-	Low
Voice CC (VCC)	-	-	-	Medium	-	-	High
⊟ Ranking (RK)	false	false	false	true	false	false	true
Default (DRANK)	-	-	-	Low	-	-	Medium
Reduced (RRANK)	-	-	-	High	-	-	Low
⊟ Security (SEC)	true	false	false	true	false	true	false
Weak security (WSEC)	Low	-	-	High	-	Low	-
Strong security (SSEC)	Medium	-	-	Low	-	High	-
⊟ Telephony (TEL)	false	true	false	true	false	false	false
TAPI (TAPI)	-	Low	-	High	-	-	-
Skype (Skype)	-	-	-	Medium	-	-	-
Jahjah (JAH)	-	Medium	-	Low	-	-	-
GSM (GSM)	-	High	-	High	-	-	-
⊟ User Interface (UI)	true	false	true	false	false	false	true
Smart Phone (SPUI)	Medium	-	Medium	-	-	-	Low
Rich Ajax Interface (RAI)	High	-	High	-	-	-	Low
Voice Control (VUI)	Low	-	Low	-	-	-	High
⊟ Map Service (MAP)	true	true	false	false	true	false	false
Google maps (GMAP)	Low	Low	-	-	Very Low	-	-
Yellowmap (YMAP)	Medium	Medium	-	-	High	-	-
⊟ Video On Demand (VOD)	true	false	false	true	false	false	false
Youtube (GVOD)	High	-	-	Medium	-	-	-
Dailymotion (DVOD)	High	-	-	Low	-	-	-
⊟ Cache (CACH)	false	false	false	false	false	false	false
DefaultCache (DCACH)	-	-	-	-	-	-	-

Figure 3.9: Impacts of variants on QoS properties

3.6 Architecture Metamodel

This metamodel (Figure 3.11) allows the designer to describe component-based architectures. At design-time, we can use any metamodel, such as the UML 2 component diagrams or SCA[1] (Software Component Architecture) to describe architectures. In this thesis we focus the scope of dynamic adaptations on reconfigurations *i.e.*, dynamic manipulation of components and bindings. These concepts are present in the above mentioned metamodels.

At runtime, we use our own core metamodel for describing architectures, called ART for (models) **At RunTime**. This core metamodel only contains the concepts and relationships we need to describe the architecture of a running system. This way, we can reduce the memory overhead caused by maintaining a model at runtime. We defined this metamodel by analyzing different existing metamodels (UML, SCA, etc) and platforms (Fractal, OpenCOM, OSGi, etc) and by focusing on our need of dynamic adaptation (reconfiguration). Since the concepts of UML or SCA and those defined in ART are very

---

[1] http://www.eclipse.org/stp/sca/

	Name	ID	context	Power Consumption	Cost	Usability	Performances	Proximity Search	Security	Disturbance
Rule	On mobile	MOB	DEVTYPE = MOB	High	-	Low	-	-	-	-
Rule	On PDA	PDA	DEVTYPE = PDA	Medium	-	Medium	-	-	-	-
Rule	On Laptop	LAP	DEVTYPE = LAP	-	-	High	-	-	-	-
Rule	General		MODE = ON or MODE = BUSY or MODE = DRIV	-	Medium	-	High	-	High	-
Rule	Voice Prefered		MODE = DRIV	-	-	-	-	-	-	Low
Rule	Text Prefered		MODE = BUSY	-	-	-	-	-	-	High
Rule	Sparetime	SPAR	MODE=FREE and (DEVTYPE = LAP or DEVTYPE = PDA)	-	High	-	Low	High	Low	Low
Rule	Offline		MODE = OFF	-	-	-	-	-	-	High

Figure 3.10: Context-dependent Optimization Rules

often in a one-to-one mapping, it is rather straightforward to implement a bi-directional model transformation.

Our generic metamodel is illustrated in Figure 3.11. A component type contains some ports. Each port has a UML-like cardinality (upper and lower bounds) indicating if the port is optional (lowerBound = 0) or mandatory (lowerBound > 0). It also indicates if the port only allows single bindings (upperBound = 1) or multiple bindings (upperBound > 1). A port also declares a role (client or server) and is associated to a service. A service encapsulates some operations, defined by a name, a return type and some parameters, similarly to a Java interface. A component has a type and a state (ON/OFF), specifying whether the component is started or stopped. It can be bound to other instances by a transmission binding, linking a provided service (server port) to a required service (client port). A composite instance can additionally declare sub-instances and delegation bindings. A delegation binding specifies that a service from a sub-component is exported by the composite instance.

Appendix A.1 shows how we can map the SCA metamodel to the ART metamodel in order to transform SCA models to ART models.

The architecture metamodel allows architects to specify their system at a rather low level of abstraction, making it possible to fully automate the reconfiguration process (see Section 5). However, in the context of large adaptive systems, it is not realistic to fully specify the architecture of all the possible configurations. Rather, we propose to rely on Aspect-Oriented Modeling techniques to automatically derive these configuration, when needed, by weaving fragments of architecture (aspects) that refine the variant of the variability model.

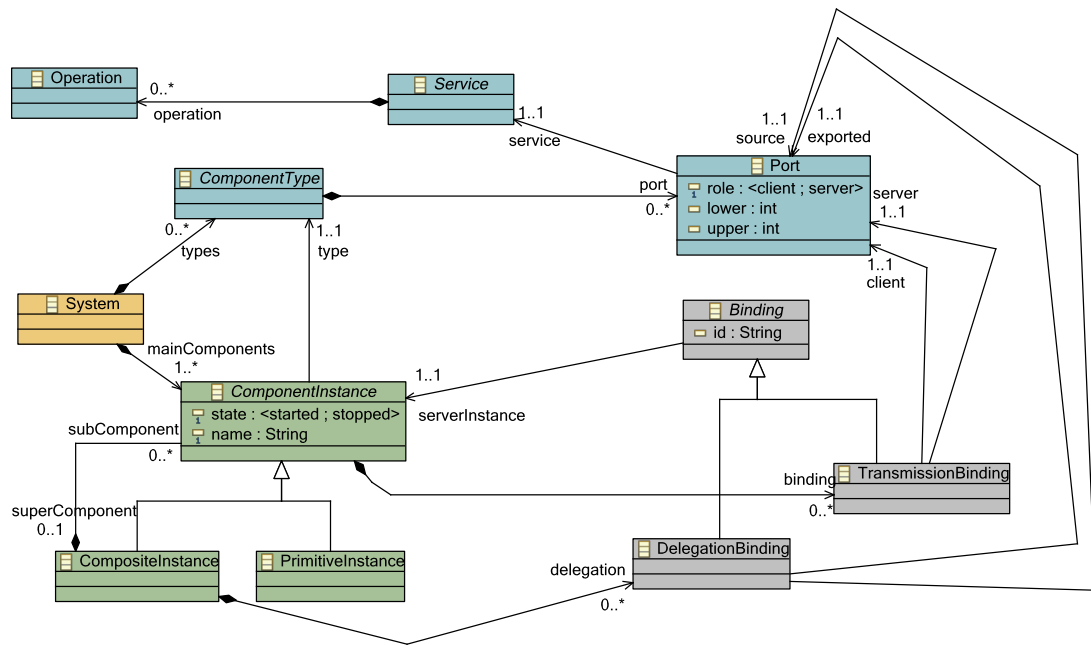Figure 3.11: Architecture Metamodel (main concepts)

Chapter 4

Aspect-Oriented Modeling to Support (Dynamic) Software Product Lines

Contents

This Chapter presents how we leverage Aspect-Oriented Modeling (AOM) to derive configurations (architectural models) from the variability model, as illustrated in Figure 4.1. It first presents the requirements that an AOM approach should provide so that

it could actually be used at runtime. We then present in details our SmartAdapters AOM approach we integrate at runtime (and also at design-time) to automatically derive configurations.

4.1   Requirements for an AOM approach usable at Runtime

This section presents the requirements for an AOM approach to be integrated at runtime (and also at design-time) to derive configurations.

- **Adoption**: The AOM language should be close to the domain-specific modeling language. The pointcut, advice and composition languages provided by the AOM approach should directly manipulate domain concepts. In our case, the DSML is the architectural metamodel presented in Chapter 3.

- **Expressiveness**: It should allow designers to define aspects that produce a result they expect. In particular, it should be able to compose cross-cutting and non cross-cutting features.

- **Performances**: It should be able to weave aspects within reasonable time and memory constraints.

- **Agility**: It should be able to adapt to the changes of the domain metamodel.

- **Tool-support**: It should be implemented, available and compatible with EMF, which is a very common modeling framework to manipulate models within the Eclipse world.

AOM weavers have proved their utility in the context of "classic" SPL to refine features [88, 77, 64, 102, 100, 120]. We claim that AOM is also a good candidate in the context of DSPL to derive configurations. However, AOM weavers are usually employed at design-time, where performances (time, memory) are not critical. This requirement is much more critical at runtime. But a more fundamental problem is the lack of available implementations. While the literature is quite prolific about AOM approaches [9, 88, 102, 78, 64], few implementations are actually publicly available and maintained. Noticeable exceptions are the weavers developed in Kermeta [112] by the Triskell team[1]. The expressiveness of state-of-the-art AOM weavers can also be discussed. Merge-based approaches [49, 130, 55] are well suited to weave non cross-cutting features but cannot really be used to weave cross-cutting features because of the lack of quantification (wildcards) mechanisms. Pointcut-based approaches are supposed to fill this lack. However, it appears that these approaches do not offer the right mechanisms to properly weave aspects,

---

[1]http://www.irisa.fr/triskell/home_html-en

as discussed in this Chapter. The agility requirement is important in the context of a research project when the metamodel often evolves. This is also important in production to support the future evolution of the metamodel to keep it aligned with the evolution of the execution platforms.

4.2 Overview

This section gives and overview on how we leverage Aspect-Oriented Modeling (AOM) to derive configurations (architectural models) from the variability model, as illustrated in Figure 4.1. AOM allows designers to refine features in a declarative way, and most of the complexity of the actual weaving/composition is managed by the weavers. This prevents designers from writing low-level and error-prone reconfiguration scripts. Moreover, AOM relies on a strong theoretical background, such as the graph theory [24, 150], and offer a good basis for early validation.
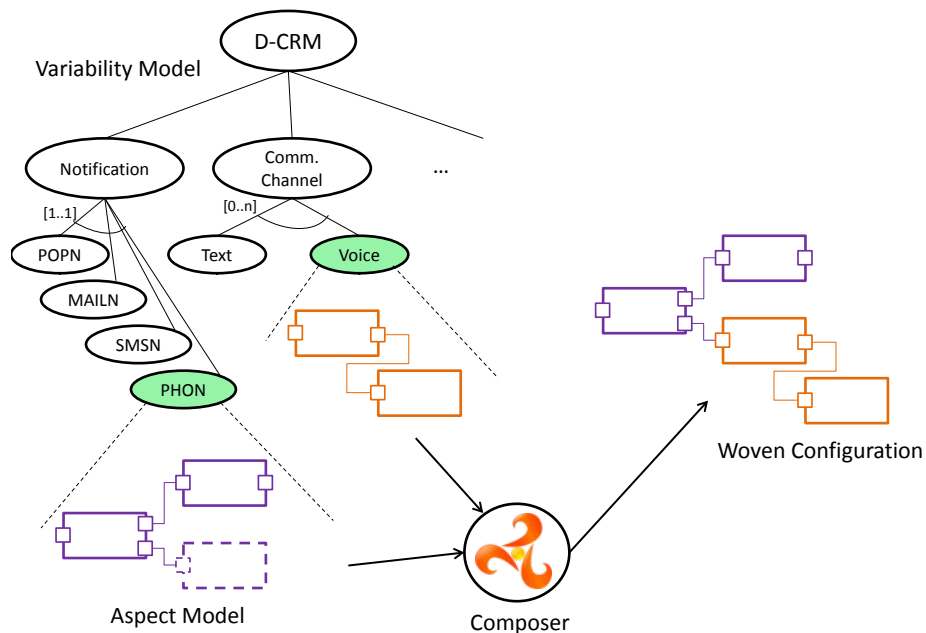
Figure 4.1: Refining features into Aspect Models and Derivation Process

It is important to note that we do not impose any particular AOM weaver. Indeed, the next step of the approach (see Section 5) makes no assumption on the way models are build. In other words, it would be possible to use any AOM weaver (or any model transformation language such as Kermeta [112] or ATL [79], or any graphical editors) to

produce these models. In the remainder of this section, we present how we leverage the SmartAdapters approach we developed in the INRIA Triskell group.

4.3   Rapid Background on SmartAdapters

SmartAdapters has been applied to Java programs [89] and UML class diagrams [88]. In these versions, the join points (the places where the aspect is woven) were manually specified. More recently, we have generalized this approach to any domain specific modeling language [101, 100] and extended the approach with a pattern matching engine to automatically detect join points [128]. This allows us to leverage the notion of aspect for any DSML [129], and especially for runtime models representing at a high level of abstraction the architecture of a system at runtime. SmartAdapters automatically generates an extensible Aspect-Oriented Modeling framework specific to our meta-model.

In SmartAdapters, an aspect is composed of three parts:

1. an advice model, representing **what** we want to weave,

2. a pointcut model, representing **where** we want to weave the aspect and

3. weaving directives specifying **how** to weave the advice model at the join points matching the pointcut model.

These concepts are integrated in a metamodel, which allows designers to instantiate aspect models. The metamodel of SmartAdapters is illustrated in Figure 4.2. Both the advice and the pointcut models allow the designer to define the aspect in a declarative way (*what*), whereas the weaving directives allow the designer to specify the composition using a simple statically-typed imperative (*how*) language.

4.4   A (not so) Simple Cache Aspect

We propose to illustrate SmartAdapters on several versions of the cache aspect of the D-CRM, to emphasize the lacks of current approaches.

4.4.1   A Naive Cache Aspect

Figure 4.3 illustrates the first version of the cache aspect, which basically consists in linking any component (in dashed line in the figure) that requires the cache service to a cache component actually providing this service.

If we apply this aspect on a simple flat architecture, the cache component is duplicated for each join point *i.e.,* each component that requires the cache service is now connected to its own cache component, as illustrated in Figure 4.4.
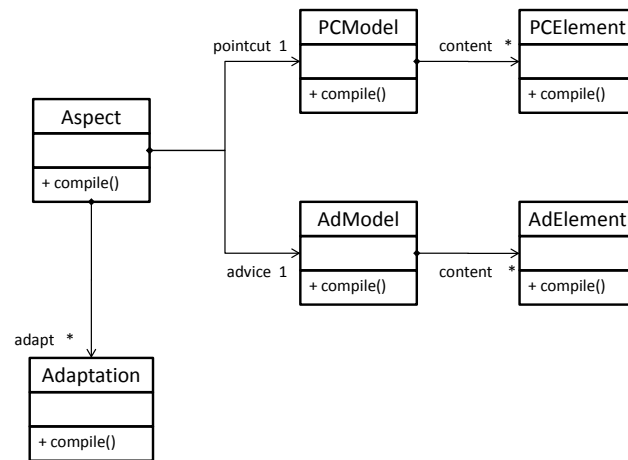
Figure 4.2: SmartAdapters Core Metamodel
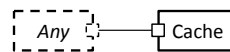
Figure 4.3: A Simple Cache Aspect

4.4.2 On the need of Advice Sharing

The previous result (Figure 4.4) is probably not the one we would expect. Rather, we would like to obtain the result illustrated in Figure 4.5. In this case, all the components that require the cache service are connected to the same cache component. In other word, the binding defined in the aspect is duplicated *per join point*, whereas the cache component is *unique*. We introduced the original notion of uniqueness in AOM in 2007, for the SmartAdapters approach [101]. In 2009, Grønmo *et al.* [65] introduce a collection operator for graph transformation, which is different from the notion of uniqueness, but which could be useful to realize uniqueness.

4.4.3 On the need of Scoped Advice Sharing

Let us now consider an architecture composed of a hierarchy of components, such as the one illustrated in Figure 4.6.

In this case, we would like each composite component to contain a cache component, which would be used by all the internal components of the composite. If we do not declare the cache component as unique, we end up we a similar problem as Figure 4.4: this component is duplicated for each join point. If we declare the cache component as unique,
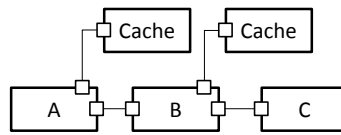
Figure 4.4: Weaving the Simple Cache Aspect
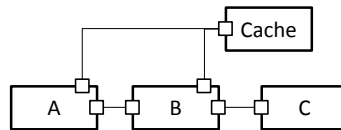
Figure 4.5: Weaving the Simple Cache Aspect: Expected result.

we end up with another problem: since a single model element cannot be contained by multiple containers (the sub-component relationship is a composite reference in the ART metamodel), it is not possible to add the same cache component into different composite components. More precisely, the cache component will be added into the first composite component and bound to all the internal components. Next, it will be moved inside the second composite component, etc. At the end, this would lead to an erroneous configuration, with bindings that "cut across" the boundaries of their composite components, as illustrated in Figure 4.7.

There is a real need for a more customizable notion of uniqueness. Note that these problems are common to all the pointcut-based AOM weavers, such as MATA [162, 77]. We thus propose the notion of scoped uniqueness to cope with this issue that we describe further in this section. Using this novel notion, it would be possible to obtain the result illustrated in Figure 4.8.

4.5 SmartAdapters: Concepts in details

4.5.1 Leveraging model typing to design advice and pointcut model

In Aspect-Oriented Programming (AOP) languages such as AspectJ [83, 84], an advice is defined as a piece of code woven at some well identified places (joint points) matching the pointcut of the aspect. The pointcut is a predicate (possibly with wildcards) over a program that may not be fully specified in order to allow quantification *e.g.*, intercept all the calls to any method of a given class.

Unlike AspectJ, which is dedicated to the Java programming language, SmartAdapters is a generic AOM approach that can be used to weave aspects into different Domain-
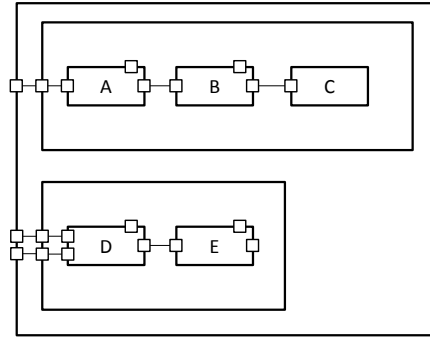
Figure 4.6: A composite Architecture
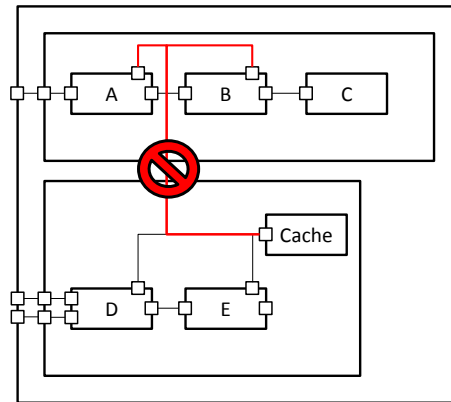
Figure 4.7: A composite Architecture, with the logging aspect badly woven.

Specific Modeling Languages (DSML). For each DSML, described by a domain metamodel (MM), SmartAdapters automatically generates two super model types, using a model transformation written in Kermeta, which allow users to design advice and pointcut models in a more flexible way. These metamodels define a hierarchy of model types [145, 144], as illustrated in Figure 4.9:

- MM is the domain metamodel. It describes the concepts of the domain, their relationships (references, inheritance, etc.) and several constraints (cardinalities, OCL constraints, etc). Designers use MM to design base models. By analogy with AspectJ, MM would represent the Java programming language. In our case, MM is the architecture metamodel presented in Section 3.

- MM' is a super type of MM. It contains all the concepts and all the relationships
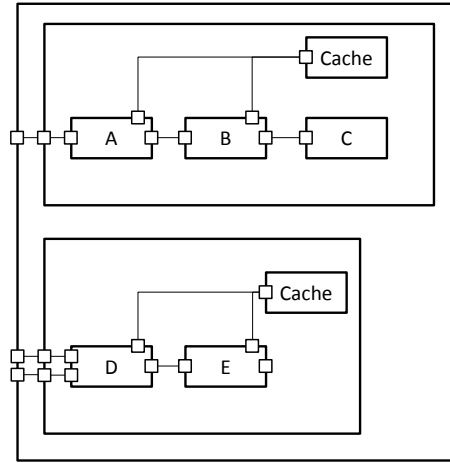
Figure 4.8: A composite Architecture, with the logging aspect correctly woven.

defined in MM, with no constraints. The invariants defined in MM are no more considered in MM'. Especially, all the lower bounds of the cardinalities are set to zero. Designers use MM' to design advice models, which are model fragments that may not ensure all the constraints defined in the former metamodel.

- $MM'_{polymorphic}$ is a super type of MM'. It is similar to MM' but has no abstract meta-class. Designers use $MM'_{polymorphic}$ to design pointcut models, which are polymorphic model fragments. This way, a pointcut can for example instantiate a *ComponentInstance* (which is abstract in MM), to specify that we want to match either a *PrimitiveInstance* or a *CompositeInstance*.

In the case of an evolution of MM, we can distinguish two cases:

- **Pure Extension**: MM is extended with new meta-classes and new relationships. In MM', this extension is optional (since all the lower bounds are set to zero). This makes it possible to load existing aspect models (conforming to the initial meta-model MM) with the extended metamodel MM.

- **Other Evolutions** may break the conformance relationship between existing aspect models and the evolved metamodel MM'. However, it is still possible to (manually) implement a model transformation that ensures the conversion between the former and the new metamodels.

Domain Metamodel, with no constraint and no abstract meta-class — MM' polymorphic

Domain Metamodel, with no constraint — MM'

Domain Metamodel — MM
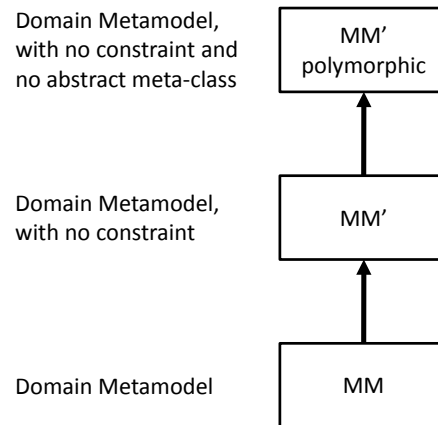
Figure 4.9: Hierarchy of model types handled by SmartAdapters

4.5.2   Defining Sharing Strategies for Advice Models

In [101] we introduced the original notion of advice sharing in SmartAdapters. This allows the designer to specify which elements from the advice are kept for all the join points and which element are instantiated for each join point match. The notion of advice sharing allows reusing some elements of an advice for each composition of an aspect in the same base model. However, specifying that a model element is shared or not is not always fine grained enough, especially in hierarchical models: state charts with composite states, component diagrams with composite components, class diagrams with packages and sub-packages, etc.

We propose the notion of scoped sharing to tackle this issue [108]. Advice models conform to the relaxed metamodel MM'. Additionally, we propose to associate a strategy to advice model elements, as illustrated in Figure 4.10. In the case there exist several join points in the base model that match the pointcut, the designer can adopt several strategies, for each individual model element of the advice:

- **Global**. A single instance (clone) of the element is introduced into the base model. In the case where there are several join points, global elements will be reused. See for example the log component in Figure 4.5.

- **Per Join Point**. A new instance of the element is introduced for each join point. See for example the bindings in Figure 4.4.

- **Unique with scope**. A single instance of the element is introduced for each zone of the base model identified by the scope. See for example the log component in Figure 4.8.
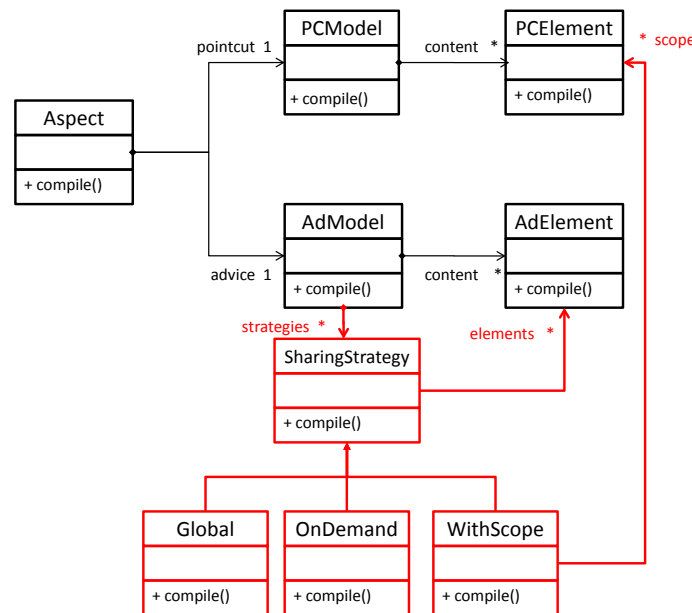
Figure 4.10: Extension of the SmartAdapters to Handle Advice Sharing Strategies

We define the scope of an advice as a set of (references to) pointcut model elements, as illustrated in Figure 4.11. In this new version of the cache aspect, we have also considered in the pointcut the composite component containing the component that requires the cache service. This composite allows us to define the scope of the cache component. Two join point points have the same scope if the pointcut model elements defined in the scope are bound to the same base model elements. In the base model illustrated in Figure 4.6, there is 4 join points: one per component that requires the cache service, as shown in Table 4.1. These 4 join points can be split into 2 scopes (defined by the 2 composite components).
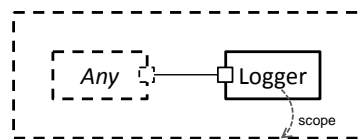
Figure 4.11: Logging Aspect revisited with a scoping strategy

The only scope we set by default in the context of architectural models is that bindings (defined in the advice) are local to the components (defined in the pointcut) they are con-

Pointcut	JP1	JP2	JP3	JP4
Any	A	B	D	E
Composite	Composite1	Composite1	Composite2	Composite2

Table 4.1: 4 Join points defining two scopes.

nected to. This default scope is particularly useful in the case of overlapping join points, as illustrated by Figure 4.12. Let us consider an aspect that binds the unique *cpt* component to any pair of already connected component (*any* et *another*), as shows in the top part of the Figure. In the simple base model illustrated in the left part of the Figure, the pointcut can match 2 times: A → B and A → C. In the case the bindings were *per join point*, this would lead to the first woven model illustrated in right part of the Figure, where A is connected twice to *cpt*. Obviously, the expected result is the one illustrated in the bottom right part of the Figure, where *cpt* is connected once to all the base components. This result can be achieved thanks to the default scope of the binding.
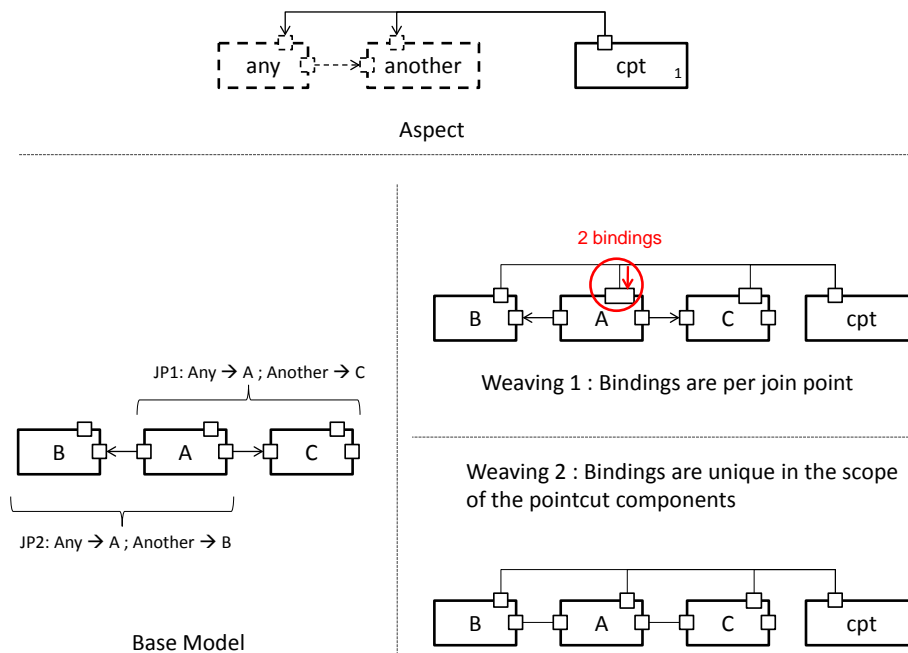
Figure 4.12: Overlapping join points: How advice sharing and set matching can help

4.5.3   Extension of the SmartAdapters metamodel

The SmartAdapters metamodel (Figure 4.13) is automatically extended via well-identified extension points (*PCElement*, *AdElement* and *Adaptation*) to provide an operational AOM framework for a given domain metamodel. We have specialized SmartAdapters for the architecture metamodel, previously described in this thesis (Section 3).

1. **PCElement**: represents an abstraction of any model element conforming to $MM'_{polymorphic}$. *PCElement* is automatically introduced as the root meta-class of all the element of $MM'_{polymorphic}$, when we specialize the framework for a given domain. The code of the *compile* method (see next sub-section) is also fully generated for all the meta-classes of $MM'_{polymorphic}$.

2. **AdElement**: represents an abstraction of any model element conforming to MM'. *AdElement* is automatically introduced as the root meta-class of all the element of MM', when we specialize the framework for a given domain. The code of the *compile* method (see Section 4.6.2) is also fully generated for all the meta-classes of MM'.

3. **Adaptation**: represents an abstraction of any domain specific weaving operation. All the domain-specific adaptations must extend this meta-class, declare some attributes (the parameters of the adaptations), and implement the *compile* method (see Section 4.6.2). We automatically generate some basic adaptations, but designers can create some additional adaptations that extends Adaptation, or modify existing ones.

For each meta-class `X` of a metamodel MM, we generate four adaptations:

1. `SetX`: this adaptation allows user to set or update (addition) any property of `X`. For example, `SetComponentInstance` allows designers to add bindings into a (client) component instance.

2. `UnsetX`: this adaptation allows user to unset or update (removal) any property of `X`. Similarly, `UnsetComponentInstance` allows designers to remove bindings from a (client) component.

3. `CreateX`: this adaptation allows user to create a new instance of `X`. It is generated only if `X` is concrete. For example, `CreateTransmissionBinding` allows designers to create a new binding, that can be manipulated in the remainder of the composition protocol.

4. `CloneX`: this adaptation allows user to clone an existing instance of `X`. It is generated only if `X` is concrete. Similarly, `CloneTransmissionBinding` allows designers to clone an existing binding, and manipulate it.
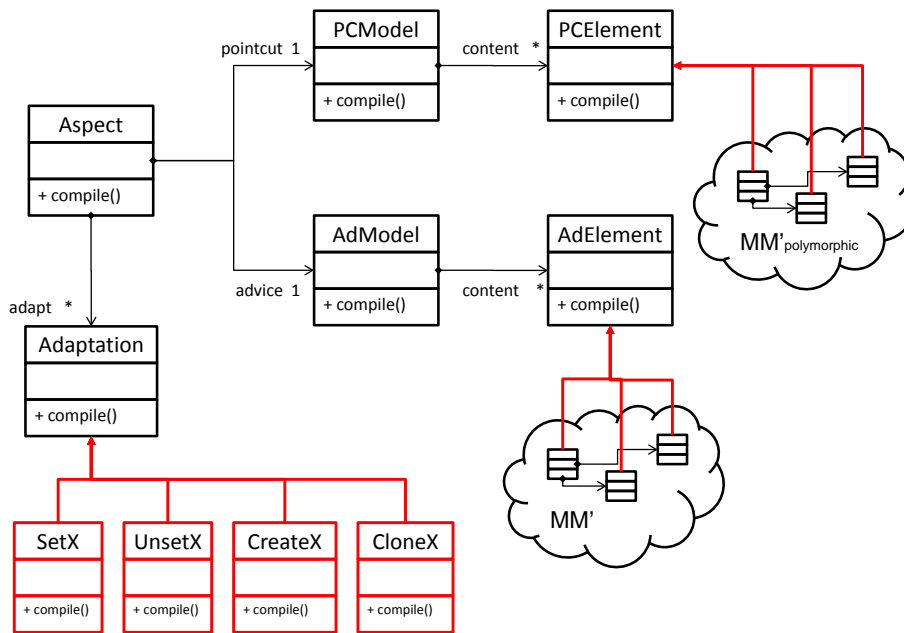
Figure 4.13: The metamodel of SmartAdapters

All the Kermeta code corresponding to the *compile* methods is fully generated. This hierarchy of adaptations is totally customizable: designers can add, remove, modify or create adaptations. The `Adapter` and `Adaptation` meta-class follow the Command designpattern. This way, the weaver can handle any sub-class of the `Adaptation` meta-class in a seamless way.

4.6 Tool Support

4.6.1 SmartAdapters V1: A Proof-of-Concept to Assess AOM to Compose Dynamic Features

The first version of SmartAdapters was developed using the Kermeta interpreter. The join point identification engine [128] was totally delegated to a Prolog engine [146], via a Java layer. While the bridge between the Kermeta interpreter and Java is seamless, the bridge between Java and Prolog is more difficult to achieve. To bridge this gap we needed to generate text files in Kermeta, that can be loaded by Prolog to:

- Initialize its knowledge base, using a textual representation of all the facts describing a base model and its metamodel (ART in our case).

- Execute a query over over the knowledge base, using a textual representation of all the variables and constraints defining the query (*i.e.* the pointcut).

The results of a query were directly pushed into the Kermeta interpreter stack so that an interpreted Kermeta program could directly use these results. However, this architecture is not realistic in the context of *models@runtime*. Indeed, this architecture has a significant memory overhead, with performance penalties due to the generation/load of text files, as illustrated by Figure 4.14. Note that MATA [78] also bi-directionally maps aspect models to another formalisms (graphs, using AGG [150]), with performance/memory penalties.
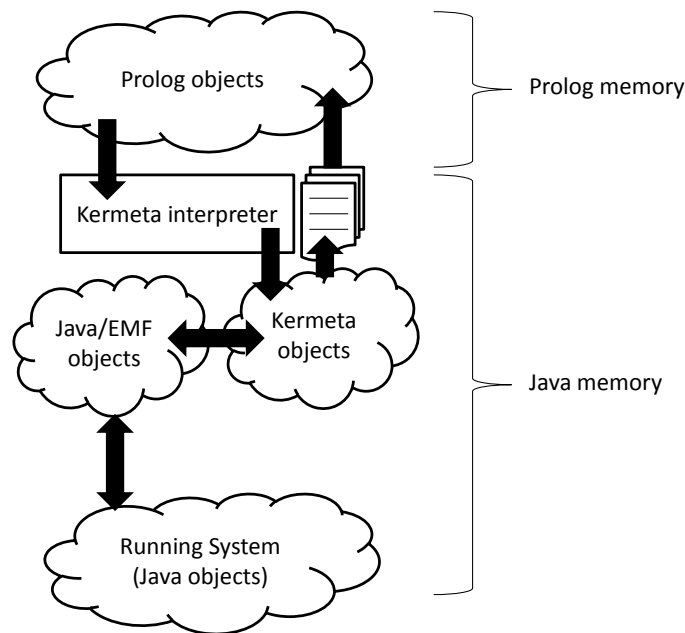
Figure 4.14: Conceptual Architecture of SmartAdapters V1

4.6.2   SmartAdapters V2: A Generative Approach to More Efficient Aspect Model Weaving

Based on these requirements, we develop a new version of SmartAdapters that can be employed both at runtime and at design-time. We rely on Drools Expert[2] (*a.k.a* JBoss Rules) to realize the join point identification step. Drools implements the Rete algorithm [54] with specific optimizations that leverage Object-Oriented concepts. In particular, Drools

---

[2]http://www.jboss.org/drools/drools-expert.html

can seamlessly apply rules on any Java program whose classes respect standard POJO conventions (private attributes with getters and setters respecting a naming conventions), which is the case of EMF. To meet the agility requirement, we rely on generative techniques. The overall approach can be decomposed into two steps:

- **At design-time** (Figure 4.15): A Meta-code generator, written in Kermeta and KET (Kermeta Emitter Template) using MOF concepts (M3 level), takes a metamodel as input and generate a metamodel-specific code generator. In our case, the metamodel is the architecture metamodel presented in Section 3 (Figure 3.11). This generated code generator uses the concepts of the domain metamodel (M2 level). It takes an aspect model as input and compiles it into Drools and Java/EMF code. Some implementation details are presented in Appendix A.2.

- **At runtime** (Figure 4.16): The runtime weaver simply takes the Drools files generated at design-time and directly process them in memory, on a base model. More precisely, the Drools scripts manipulate the base model to obtain the woven.
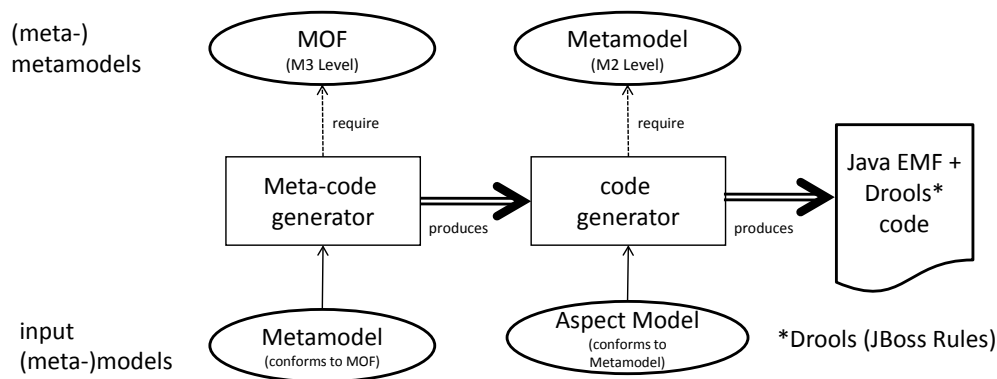
Figure 4.15: SmartAdapters V2, at design-time

The meta-code generator takes a metamodel as input (M2 level) and weaves a 4-pass visitor (and its Kermeta implementation) into this metamodel. The first pass is in charge of appending a sequence of Drools statements corresponding to the pointcut model (see the *when* clause in the script below). The second pass is in charge of appending *create** statements corresponding to the creation of the advice model using the EMF API of the metamodel. In particular, it takes care of the uniqueness of the advice elements. The third pass in responsible for appending *set** statements which link together advice model elements. Finally, the forth pass is in charge of appending the code related to the composition protocol of the aspect to link the advice to the pointcut (*i.e.* to the places of the base model that match the pointcut model).
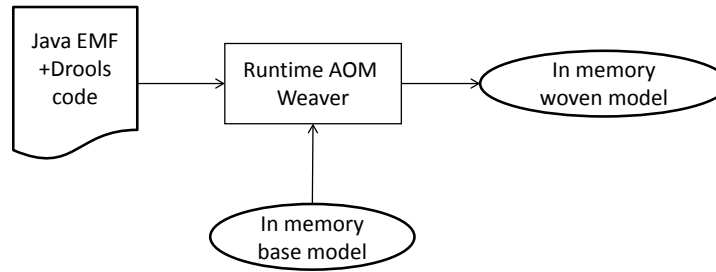
Figure 4.16: SmartAdapters V2, at runtime

The Java/Drools code associated with a simple logging aspect (similar to the cache aspect presented in this Chapter) is presented and discussed in Appendix A.3 (logger is unique) and Appendix A.4 (logger is unique within the scope of a composite).

This section showed how we use the SmartAdapters approach to refine and compose the dynamic features of a DAS. It is important to note that the remainder of our approach is not specific to SmartAdapters. In other words, it is possible to use any weaver to produce configurations. The next chapter explains how we leverage woven configurations to automatically drive the reconfiguration process.

4.7   Discussion

The first version of SmartAdapters was developed at the very beginning of this thesis. The goal of this proof-of-concept prototype was to assess the utility and the usability of AOM in the context of adaptive systems. The focus was thus on the adoption, expressiveness, tool-support and agility requirements:

- **Adoption**: SmartAdapters directly manipulates the concepts of the ART metamodel (but it is possible to specialize SmartAdapters for any other domain metamodel), which is inspired by UML component models and well known platforms like Fractal or OpenCOM. These concepts are easily understood by designers and architects. Especially, the pointcut language is directly derived from the ART metamodel. There is no need to learn a new language for describing pointcuts. Instead, a pointcut is any fragment of architecture. Similarly, the advice is also a fragment of architecture. In addition, SmartAdapters requires designers to implement a composition protocol, describing how the advice is woven into the pointcut, quite similarly to MATA. This language is a dedicated imperative and statically-typed language that can easily be manipulated by people with a programming experience, *e.g.* in Java. However, the very bad performances were making the iterative process of the aspect design longer.

- **Expressiveness**: The pointcut language allows defining cross-cutting and non-cross-cutting aspects. If the pointcut is very precise, it will match a very reduced set of join points. If the fragment of architecture describing the pointcut is more "vague", it will possibly match a wide set of join points. The notion of advice sharing is particularly useful to properly manage the advice in the case there are several join points.

- **Tool-support**: The first version of SmartAdapters was implemented in Kermeta and used a Prolog back-end for the joint point detection. It was able to weave all the aspects defined by the different users of SmartAdapters. The second version of SmartAdapters is also implemented as a Kermeta program, which generates code. This generated code contains all the information for detecting join points (Drools) and for weaving the aspect (Java/EMF).

- **Agility**: SmartAdapters is based on generative techniques. If the metamodel evolves, designers simply have to regenerate the weaver. If the metamodel is evolved by a pure extension, it is possible to ensure the retro-compatibility of aspect models, since this evolution would not break the conformance relationship. In the case of other modifications of the metamodel, it is not possible to automatically ensure the retro-compatibility. This problem is off course not specific to SmartAdapters, and all the tools based on a metamodel suffer from this problem.

SmartAdapters V2 has the same characteristics than the initial version. However, the performances have significantly been improved, as detailed in the validation chapter (Chapter 7).

Chapter 5

Synchronizing the runtime with design-time models

Contents

This Chapter presents how we maintain an architectural model in a causal connection with the running system, as well as a context model, in a one-way direction (from the runtime to the model).

5.1 Requirements for an "intelligent" Reflection Model

This section gives some requirements for a more "intelligent" reflection model, as we discussed in the Introduction chapter of this thesis (Sections 1.2.1 and 1.2.4).

- The reflection model should be more independent from the reality, in the sense that it should avoid "*the complexity, danger and irreversibility of reality*".

- However, it should be totally seamless to re-synchronize the reflection model with the reality *i.e.*, to actually adapt the running system.

- The link between the reflection model and the running system should not be platform-specific.

We claim that an explicit and more independent reflection model is one of the keys to tame complex dynamic systems. It would allow reflection to evolve from a powerful but hazardous "*what happens when I do...*" process to a powerful and controlled "*what would happen if I would do...*" process. Indeed, the idea of the "models@runtime" community is to leverage MDE techniques and tools at runtime to raise the level of abstraction. At the model level, it does not make sense to restrict the expressiveness of a transformation language to force designers to respect the constraints at all the steps of a transformation, or to improve the complexity of the composition tools to respect these constraints, since a model "*allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality*" (See Rothenbers's citation in the Introduction of this thesis). This would make the modeling space much more closer to the real world, minimizing the impact and the benefits of Models@Run.Time [114, 18]. The ultimate goal of models@runtime is to raise the level of abstraction of running (adaptive) systems, not to restrict existing MDE tools, techniques and methodologies to fit the constraints of such system. We argue that a more independent reflection model would allows us to raise the level of abstraction of models@runtime and to keep modeling tools expressive. Indeed, if the running system is adapted while the model manipulated, this could make the system inconsistent because the model is temporarily inconsistent.

5.2   Overview

In this section, we give a rapid overview on how we synchronize the runtime with design-time models, as illustrated in Figure 5.1.

To safely adapt a running system and take suitable decisions, it is important to have a well-fitted view of the context, which should be sufficiently detailed to reflect the reality, and sufficiently abstract to enable efficient reasoning. The context model is really an abstraction of the context, and cannot be employed to directly act on the reality. Instead, changes in the context can trigger a reasoning process (adaptation rules) that can finally act on the system.

To realize the causal connection between the architectural model and the runtime, the basic idea is to generate reconfiguration scripts on the fly, which allow to make the system to switch from its current configuration to a newly produced and independent one. This

Figure 5.1: Design models at runtime: Online and Offline Activities

way, we prevent architects from writing numerous low-level, platform-specific and error-prone scripts.

5.3 Step-wise abstraction of the execution context

In this section, we present a flexible way to progressively fill the gap between low-level probes integrated into the runtime and a high-level context model defined at design-time. This high-level model serves as a basis for reasoning about the context and driving the dynamic adaptation process. By abstracting the context and reducing the number of variables and values, it is possible to reason more efficiently [51, 37]. However, the gap between the real execution context and design-model specify the context is wide. On the one hand, probes integrated into the runtime generate a (quasi-) continuous flow of quantitative raw data. On the other hand, designers usually employs qualitative value to describe the context at a high level of abstraction. Several techniques already exists to bridge this gap: Complex-Event-Processing (CEP) [93], fuzzy-logic [33], etc. Instead of defining complex transformations or queries that directly link low-level events to high-level concepts defined in the context model, we propose to abstract the context in a step-wise manner, *à*

la MDE.

A classic MDE process generally consists in refining step-by-step high-level models into lower level abstractions. When the gap between to levels of abstractions is large, *e.g.* between requirement and architecture, it is often difficult to fully automate the transformation. When this gap is limited, *e.g.* between a fully specified UML model and Object-Oriented code, it is possible to fully automate the transformation. We basically propose to adopt a MDE approach, in the opposite way: from runtime to model.

5.3.1   An Overview of Complex Event Processing and WildCAT 2.0

Complex Event Processing (CEP) components, such as Esper[1], offer expressive mechanisms to handle large amount of runtime events such as pattern matching on events, time windows, aggregation functions (min, max, average, etc). Unlike hard thresholds, these queries make it very simple to deal with permanent context oscillations e.g., by defining thresholds on average values computed on a time slot. As previously described, the basic idea is to define queries over runtime events in order to update the context model.

We use WildCAT [26, 39][2], a generic open-source monitoring framework built upon Esper, for developing context-aware applications. WildCAT has formerly been implemented by the AsCoLa team and improved in the context of the GALAXY INRIA cross-project[3]. It allows monitoring large scale applications by easily organizing and accessing sensors through a hierarchical organization. This section first presents the meta-model of WildCAT to organize sensors. It then details how we use WildCAT to create and update a model of the context, in a step-wise manner.

The root concept of WildCAT is the *context*. A context is an oriented tree structure, similar to the Unix file system, which contains two types of nodes:

- Attribute, which holds some values. Attributes are the leaves of the sensors tree (like files in a file system). WildCAT proposes three kinds of attribute:

    - Basic attribute holds static values. Their values do not evolve unless programmatically modified.

    - Active attribute or POJOAttribute represents WildCAT sensors. In general, these attributes are associated to probes linked to the environment or the execution context (CPU, Memory, Thermometer, Camera, etc).

    - Synthetic attribute or Query Attribute holds the results of expressions on other attributes. It can for example aggregate and transforms the values provided by other attributes e.g., compute mean values during 10 seconds.

---

[1]`http://esper.codehaus.org/`
[2]`http://wildcat.ow2.org/`
[3]`http://galaxy.gforge.inria.fr/`

- Resource, which contains sub-resources and attributes (like folders in a file system).

  - Basic resources allow the designer to structure the monitoring model. For example, each node of a distributed system could be a basic resource, containing sub-resources related to the memory, the CPU, etc.

  - Symbolic links are special resources that refer to another resource. Symbolic links are used to create "short cuts" in the monitoring model in order to access more rapidly to the important resources or attributes, without navigating the whole tree. An OCL constraint specifies that cycles are not allowed (constraint similar to inheritance cycle in UML or Java programs).

5.3.2 Complex Event Processing to Update Context Models

Context variables defined at design-time describe the environment at a high level of abstractions. In practice, it is more likely that several (possibly heterogeneous) runtime probes are needed to compute and update a context variable. For example, to determine whether there is a fire at an airport, we would probably combine the values provided by different heat and smoke sensors.

The CEP engine (Esper) integrated in WildCAT allows expressing EQL (Event Query Language) queries to compute attributes. The EQL language has an SQL-like syntax and provide powerful aggregators and functions. It is for example very simple to compute min, max or average on time or event windows *e.g.*, the average CPU load during the last 20 seconds.

Figure 5.2 illustrates two possible ways of abstracting the actual execution context:

- **Priority to Reactivity**: When the monitored value is above a given threshold, it is abstracted as *high* in the context model. When the monitored value is below the same threshold, it is abstracted as *low* in the context model. This monitoring is very simple to implement (even without CEP) since the abstraction (*high* or *low*) is directly related to the current value of the monitored value. While this approach is very reactive and can trigger a reconfiguration with almost no delay, it is particularly unstable in the case where the real value oscillates around the threshold. This could result in many successive dynamic reconfigurations, which could potentially be enacted slower than the context changes. In the best case (reconfiguration is much more faster than the context change) the system will always be in an adapted state. In the worse case, the system will always be in a state that is not adapted to the current context. In Figure 5.2, this monitoring leads to 6 changes in the context model.

- **Priority to Stability**: There exists several ways to improve the stability of the monitoring *e.g.*, to implement an hysteresis cycle or to use CEP. In the example, we defined 2 CEP queries. The first one simply specifies that the value is considered to be

low as soon as the real value is below a given threshold. The second one specifies that the value is considered to be *high* if there exists no value below a given threshold during a given time slot. In Figure 5.2, this monitoring leads to 2 changes in the context model.
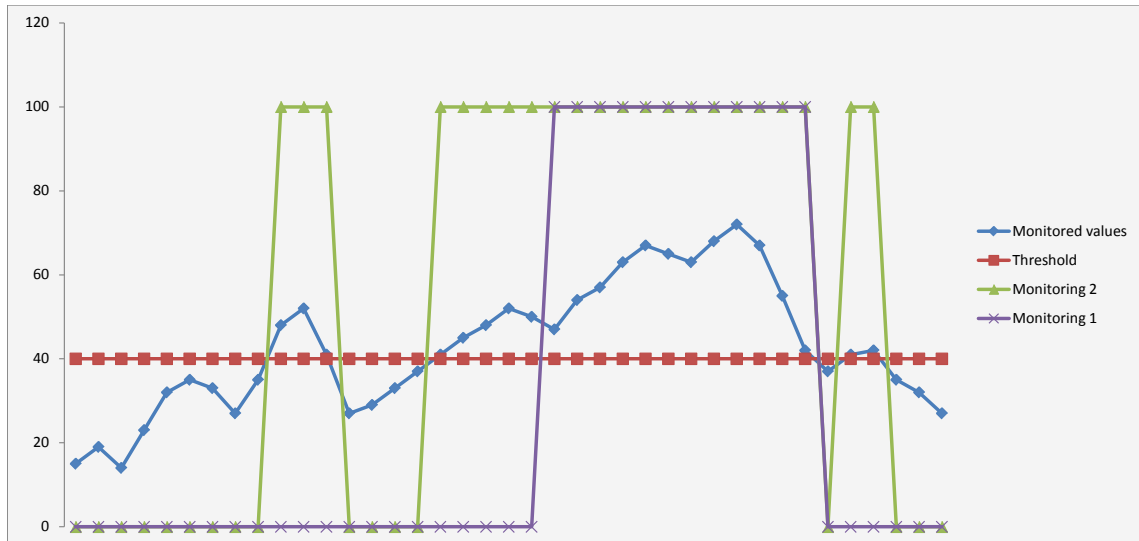
Figure 5.2: 2 Different ways of Monitoring

Complex Event Processing in just one possible way to update the context model. It is important to note that the only information we need to reason about the context is the context model itself. It is thus possible to use different techniques to update this context model, depending on the case studies.

5.3.3   WildCAT/EMF to monitor models

It is possible to use the notifications mechanism provided by EMF to be notified when a model is updated (creation, deletion or update of model elements). We leverage and extend this mechanism so that WildCAT is now able to consider the EMF notifications [109]. This way, it is possible to create intermediate context models that can be observed by WildCAT to create more abstract context models.

An interesting feature of EMF is the notification framework. EMF automatically provides notification functionality in case of changes in the model. It is possible to register observers/listeners that are notified when the model is modified. There are six types of event: *ADD*, *REMOVE*, *ADDMANY*, *REMOVEMANY*, *SET*, *UNSET*, which are respectively raised when an element is added or removed into/from the model, when a collec-

tion of elements is added or removed in/from the model or when an object property is set or unset (set to null).

Using WildCAT/EMF, it is thus possible to:

- monitor probes integrated into the runtime to build and update context models

- monitor context models to build and update other (more abstract) context models

5.4 Causal Connection between the runtime and an architectural model

Depending on the context, suitable aspect models are woven to obtain a target configuration. This section describes how we automatically generate reconfiguration scripts to adapt the running system. This automatic generation of reconfiguration scripts can be seen as a higher-order transformation, which would transform a model transformation (a sequence of aspect model weaving) into a transformation of the running system (dynamic reconfiguration), as illustrated in Figure 5.3.
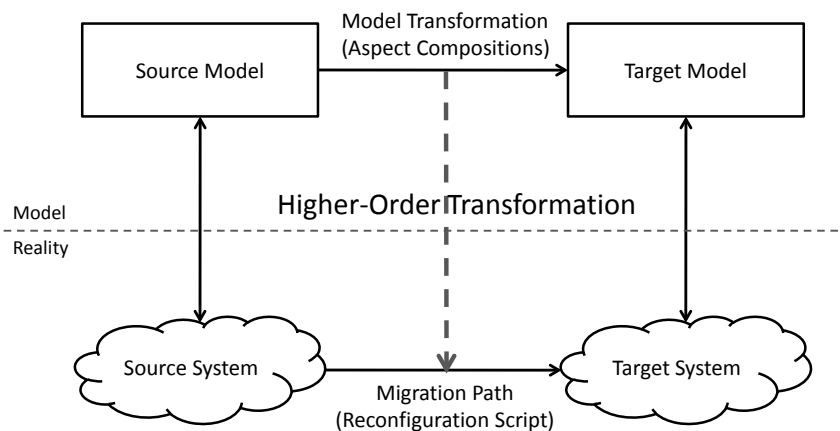
Figure 5.3: Higher-Order Transformation: From the Model to the Reality

A model transformation (see top part of Figure 5.3) usually transforms a valid model into a valid model. However, it provides few guarantees for all the intermediate steps needed to transform the model. This allows designers to implement model transformations in a more flexible way, and tools to execute these transformations in a more efficient way, without considering all the implicit and explicit constraints defined in the metamodel at all the steps of the transformation. In the case of a model composition, the

designers usually provides a set of model, and the tool is responsible for implementing and executing the model transformation making it possible to realize the composition.

A reconfiguration script (see bottom part of Figure 5.3) also transforms a valid configuration (of a running system) into a valid configuration. However, all the intermediate steps are very important, especially to correctly deal with the life-cycle of components. A reconfiguration script directly manipulates the reality (the running system), and cannot avoid the complexity, danger and irreversibility of reality: it should deal with. Writing a reconfiguration is thus intrinsically more complex than writing a model transformation, or designing an aspect model. All the actions (*e.g.,* remove component) have to be carefully ordered to realize a safe migration path [167]. Otherwise, this could lead the system into an inconsistent state. The basic idea of the higher-order transformation depicted in Figure 5.3 is to alleviate designers from writing such low-level and error-prone scripts.

Implementing this higher-order transformation as such would make this implementation very specific to the model transformation tools, aspect weavers or model composers, and how these tools actually manipulate the model. Indeed, each particular tool has its own representation of what a model transformation is. If we consider a model transformation as a black box, it can be conceptualized as:

$$MT: \{model\} \rightarrow model$$

In the case of a model transformation, MT usually transforms a single model into another model. In the case of a model composition, MT composes a set of input models into a single one. Note that the metamodel of the input models and the output model may be different. In the previous section, we use AOM techniques to compose homogeneous models, to obtain a whole configuration. However, it would be possible to use any tool able to produce architectural models. In the remainder of this section, we will explain how we can realize a technological independent higher-order transformation, by leveraging the reflection model (representing the running system) and a target model *e.g.*, which is the result of MT.

5.4.1   Maintaining a Reflection Model at Runtime: Strong Synchronization from Runtime to Model

The first step consists in maintaining an **explicit** reflection model representing the running system. This is basically the idea of computational reflection [94]. However, current adaptive platforms such as Fractal or OSGi only propose an implicit reflection model, strongly synchronized in both directions (runtime ↔ model).

In all the cases, the synchronization between the runtime and the model should be strong, so that the model is not biased and it is possible to reason on an always up-to-date reflection model. We can distinguish two main ways to realize this synchronization from the runtime to the model:

- **Pull mode**: It consists in using the reflection API provided by the platform to know the current state of the system. This API mainly gives information about the components and bindings (connection between components) composing the system, their states (started, stopped, etc), etc. This way, it is possible to build an explicit reflection model **from scratch**, every time we need to reason on the running system. Modern adaptive execution platforms, such as OSGi, OpenCOM or Fractal, propose this kind of API. However, this pull mode has a major drawback: the reflection model is build from scratch, whereas the running system evolves "continuously". Every time we need to reason, the reflection model should be entirely rebuilt even if the running has not significantly changed.

- **Push mode**: It consists in using probes integrated in the execution platform, which notify from any significant change that appears in the running system. For example, the probes should be able to notify observers that components and/or bindings have been introduced and/or removed, etc. This way, it is possible to **incrementally** build and update the reflection model. However, modern platforms only offer a limited support for monitoring the evolution of a running system. Fractal offers no probe for monitoring a running system, while OSGi offers some primitives to be notified from some events related to the running system.

5.4.2 Online Validation to Check Configurations when Needed

Online validation is important in the context of complex adaptive systems. Indeed, it is not always possible to validate the huge set of possible configurations at design-time, because of time and resource issues. However, it is not conceivable to make the system to migrate to a configuration that is not valid. This is why we validate all the produced configurations (by aspect weaving) before actually performing the reconfiguration.

The online validation of woven configuration relies on invariant checking [105]: for all the produced configurations, we check that all the invariants are ensured. We use the open-source Kermeta metamodeling language [112] to manage different checking strategies.

The Kermeta seamless weaving engine allows a designer to re-open meta-classes defined in a metamodel in order to extend them by weaving contracts (invariants, pre/post-conditions), references, attributes, super types and behavior. Here, we leverage the ability of Kermeta to extend meta-classes with OCL invariants. Each strategy is defined in a separate Kermeta aspect (represented as a layer with dashed border lines), typically:

- **Generic strategy**: Defines invariants that any architecture should ensure. For example, we check that every binding links a client port to a compatible server port.

- **Platform-specific strategy**: Defines invariants specific to a given execution platform. For example, the OSGi execution platform does not allow composite components,

whereas our metamodel allows components to be contained by composite components (not shown in the simplified version illustrated in).

- **Application-specific strategy**: Defines invariants specific to a given application. It can, for example, check the presence of some precise components in the architecture.

Finally, we can define a global checking strategy by automatically merging relevant strategies (aspects) with Kermeta. Any produced configuration will be checked against this global strategy to ensure its consistency.

5.4.3   Model Comparison to Detect Changes Between to Configurations

As explained in the previous section, we refine features using different AOM approaches, depending on their cross-cutting natures. These approaches offer high-level composition mechanisms, which were not implemented to consider runtime issues. AOM weavers do not ensure any specific order when executing atomic weaving actions. If these sequences of actions were directly reflected to the running system, this could make the running system inconsistent. It could be possible to modify the internal code of these weavers, to ensure a specific order among the atomic weaving actions. However, some of these tools are implemented with generic reflexive algorithms (*e.g.*, Kompose [49]), not easily open to modifications, while some other tools use graph transformation engine (*e.g.*, MATA [160]), with a bi-directional transformation to another formalism. Instead, and rather similarly to some other very recent models@runtime approaches [114], we propose to rely on a model comparison between the reflection model and a newly produced model. This model comparison removes all the conceptual and technical dependencies with any particular weaver, which is able to produce an ART model by weaving ART fragments.

The model comparison between two architectural models (reflection and target) is rather simple and can be expressed using the set theory, as illustrated in Figure 5.4:

- all the elements of the reflection model that have a matching counter part in the target model are kept,

- all the elements of the reflection model that have no matching counter part in the target model will be removed,

- all the elements of the target model that have no matching counter part in the reflection model will be added.

In this example, the user was previously driving and he is now in a meeting. Some components (and bindings) of the D-CRM, such as the Calendar or the Address DataBase are common to both configurations, and are consequently kept. The components and bindings specific to the *driving* context are removed and the components specific to the *meeting* context are added.
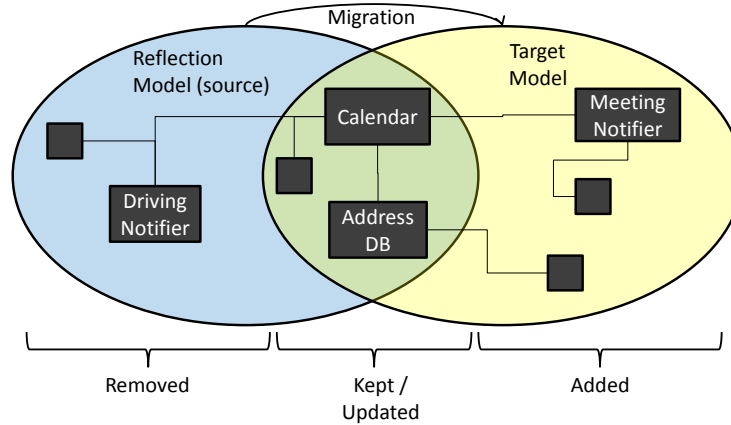
Figure 5.4: Model Comparison between the Reflection Model and a Target Model

Precise Definition of Model Comparison   We formalize the model comparison of two models conforming to the ART metamodel as a binary relationship, defined on the elements of the ART metamodel: ART $\times$ ART denoted by $\equiv$.

1. ComponentInstance c1 $\equiv$ ComponentInstance c2 $\Leftrightarrow$ **c1.container $\equiv$ c2.container** $\wedge$ c1.name = c2.name $\wedge$ c1.type $\equiv$ c2.type

2. ComponentType t1 $\equiv$ ComponentType t2 $\Leftrightarrow$ t1.name = t2.name $\wedge$ |t1.ports| = |t2.ports| $\wedge$ $\forall$ p1 $\in$ t1.ports, $\exists$ p2 $\in$ t2.ports | p1 $\equiv$ p2

3. Port p1 $\equiv$ Port p2 $\Leftrightarrow$ p1.role = p2.role $\wedge$ p1.service $\equiv$ p2.service

4. Service s1 $\equiv$ Service s2 $\Leftrightarrow$ s1.name = s2.name

5. Binding b1 $\equiv$ Binding b2 $\Leftrightarrow$ **b1.clientInstance $\equiv$ b2.clientInstance** $\wedge$ b1.serverInstance $\equiv$ b2.serverInstance $\wedge$ b1.client $\equiv$ b2.client $\wedge$ b1.server $\equiv$ b2.server

   An element *elt* should be removed iff:
   elt $\in$ *reflection model* $\wedge \neg \exists$ *elt'* $\in$ *target model* | *elt* $\equiv$ *elt'*

   An element *elt* should be added iff:
   elt $\in$ *target model* $\wedge \neg \exists$ *elt'* $\in$ *reflection model* | *elt* $\equiv$ *elt'*

   This definition is implemented in Kermeta [112], which offers built-in OCL-like operators, such as *forAll* or *exists*, and automatically compiled into Java. The ART metamodel and the associated constraints ensure that each element can be matched at most once, because of name-space conventions. In addition, the containment relationships (in bold) plays an important role for the matching of components and bindings: two components

(*resp.* two bindings) match only if the component containing them also match. We thus browse the 2 models according to the containment relationships[4].

5.4.4   On-Demand Synchronization from Model to Runtime

During the model comparison, the comparator uses an abstract factory to instantiate atomic reconfiguration commands. However, these commands are not directly executed. In other words, the running system is not adapted during the model comparison. Instead, these commands are temporarily stored and sorted, before the whole sequence of commands is actually executed. This makes it possible to use planning algorithms to sort the commands. We currently use a simple heuristic to sort the reconfiguration commands:

1. Components (that should be stopped) are stopped

2. Bindings are removed

3. Components are removed

4. Attributes (of already present components) are updated

5. Components are added (and their attributes are set)

6. Bindings are added

7. Components are (re-)started
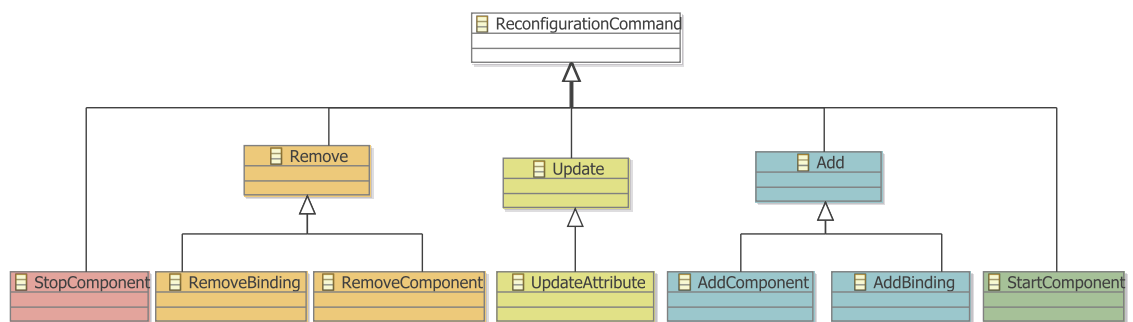
Figure 5.5: Hierarchy of Reconfiguration Commands

   In practice, each type of command is associated with a real value defining its priority. By default, the priority is a natural integer following the above enumeration. However,

---

[4]Moreover, since each model element is contained by exactly one container (except the root element) the browsing the containment graph ensures that we visit each element exactly once.

it is possible to modify this priority by adding a real value in [0..1[to the default value. This way, it is possible to finely order the commands within a given category without modifying the overall ordering.

This topological sorting ensures that the life-cycle of the components is correctly handled. The order inside a given type of commands is arbitrary, except for start and stop commands. These commands are ordered according to the client/server dependencies of components in the case where no dependency cycle exists, as illustrated in Figure 5.6. All the cases where a dependency cycle exist are forbidden (by invariant) except if there exists a weak link in the cycle and if the client components with a [0..N] cardinality (optional port) allow hot plugging their optional dependencies *i.e.* without stopping and restarting the components. If components do not allow hot plugging their optional dependencies, all the configurations with a dependency cycle should be forbidden by a invariant. Figure 5.7 illustrates a case with a dependency cycle with an optional dependency (dashed line). To reason about the start/stop order of components, we simply ignore the weak link and we thus virtually break the dependency cycle, making it possible to order the start/stop order of these components similarly to Figure 5.6.
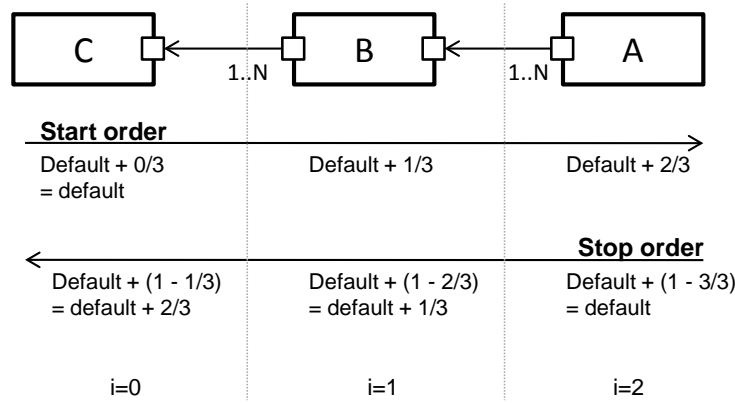
Figure 5.6: Start/Stop Order if there is no cycle

For each chain of n dependent components, the start (resp. stop) order of the ith component is computed as follows: *Default* + i/n (resp. *Default* + 1 - (i/n)).

When all the commands have be instantiated, after the model comparison finishes, the whole sequence of commands is executed. A command is only executed if the previous one correctly terminates. In the case a command encounters a problem, the reconfiguration process stops and a report is properly logged. This way, we avoid cascades or errors *e.g.*, bindings that cannot be connected because a component could not be added. Following the ideas of Léger *et al.* [91, 42], we have implemented a roll-back mechanism, which basically consists in undoing the command that crashes, as well as all the previous ones.
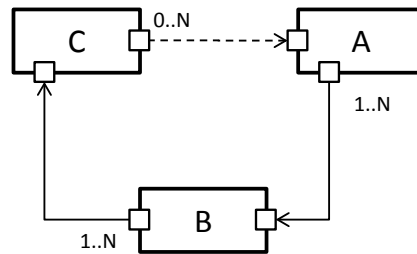
Figure 5.7: Cycle with a weak link

Each command as the following structure:

```
1   public interface ReconfigurationCommand {
2
3       //Returns the priority of the command, which determine its order of execution
4       public int getPriority();
5
6       //Changes (augment) the default priority
7       //0 <= i < 1
8       public void changePriority(int i);
9
10      //Checks the consistency of the command
11      public boolean check();
12
13      //Executes the command i.e., an atomic runtime adaptation
14      //Returns the status of the reconfiguration
15      public ReconfStatus execute();
16
17      //Acknowledge the command i.e., update the reflection model
18      //when the command successfully executes.
19      public void doAck();
20
21      //Undo the command.
22      public void undo();
23  }
```

We define the parameter of each concrete command as public attributes, which can be set after instantiating a command. For example, a RemoveComponent command has a *ComponentInstance* as parameter. The lifecycle of a command is defined as follows, in the case it successfully executes:

1. Initialization step

   (a) the command is instantiated

   (b) its parameters are set

2. Execution Step

   (a) The `check` method verifies that all the parameters are correctly set

   (b) The `execute` method realize an atomic reconfiguration and returns a status, basically fail or success

   (c) The `doAck` method updates the reflection model, in the case of a successful reconfiguration, by adding (`Add*` command), removing (`Remove*` command) or updating (`Start`, `Stop` and `Update*` commands) model elements.

5.5   Models@Runtime to Support Offline Activities

Leveraging design models at runtime alleviates designers from performing low-level and error prone tasks (*e.g.*, writing reconfiguration scripts) and allows reasoning more easily by abstracting irrelevant details of the running system and it execution environment. In addition, models@runtime [114, 18] enable performing offline activities:

Definition 5.1 Offline *activities include design-time activities as well as all the activities that could be performed while the adaptive system is running, independently of the adaptive system and without affecting the current state of the adaptive system.*

   By opposition:

Definition 5.2 Online *activities include all the activities that must be performed at runtime to adapt to the current context and/or current user need.*

   In Figure 5.1, a dashed line separates the online activities from the activities that can be performed both online and offline. At runtime (online), when the context model is updated, this triggers reasoning, which in turn can produce a new configuration (by aspect weaving) and dynamically reconfigure the running system.

   Offline activities include all the activities prior to the initial deployment of the adaptive system (requirement engineering, design, modeling, architecture, etc). They also include activities performed after the initial deployment: continuous design (redesigning the system after initial deployment), prediction (what would be the configuration of the system if the context evolves in this direction), etc that could be performed by the adaptive system itself (*e.g.* when the context is stable and if there are enough resources) or by a tiers system that has some knowledge (models) about the adaptive system. Indeed, models can easily be serialized (in XMI[5] using EMF and transmitted to tiers systems.

---

[5]XML Metadata Interchange

It thus become possible to re-work the design of the system while it is running, by modifying the variability model (by adding/removing or updating dimensions/variants or constraints) or by modifying the adaptation logic (by adding/removing or updating ECA-like rules, impacts or goals). After modifications, and after offline validation, these updated models can then be transmitted to the reasoner component deployed at runtime, so that they can reason on new inputs. In the case where new variants are added into the variability model, the component types (needed to instantiate components) that are not currently present on the platform are automatically downloaded from a repository specified in the model.

Another activity that could be performed offline is prediction. This can be achieved on the platform where the adaptive system executes, or on a tiers platform. This can be achieved by analyzing the current context model, as well as the past context models, to determine possible future contexts. For each possible future context, it is possible to determine the most adapted configuration and actually produce and validate this configuration. These possible future configurations populate a cache of configuration so that it is possible to directly use woven configuration (if the current context has changed to an anticipated context) at runtime (online) instead of performing aspect weaving and online validation. Otherwise weaving and validation can be performed online.

5.6 Discussion

In this chapter, we explained how we decouple the reflection model from the reality, and how we resynchronize the model with the reality using a model comparison. This model comparison allows us to fully generate the reconfiguration script that can make the adaptive system to switch from its current configuration to a target configuration. This way, it prevents architects from writing low-level reconfiguration scripts. It also makes the adaptation step totally independent from the way we produce the target model: it is possible to use aspect model weaving, model transformation or even to produce models by hand in a graphical editor. However, this model comparison has a cost at runtime, as discussed in Chapter 7. It would be rather straightforward to directly make SmartAdapters to instantiate (not execute, to allow online validation) the reconfiguration commands when the aspects are woven. This would make the causal faster (since no model comparison would be needed), but it would make it specific to the SmartAdapters weaver.

Chapter 6

A Model-Oriented and Configurable Architecture to Support Dynamic Variability

Contents

In this Chapter, we introduce the reference architecture we have developed in the context of the DiVA project, to support the dynamic adaptation of complex adaptive systems. Figure 6.1 illustrates a typical (data-oriented) configuration of this architecture, composed of 3 layers:

- **Platform-independent, Model-based layer**: The components defined in this level consume and produce models conforming to the metamodels presented in Section 3. It is totally independent from any execution platform. This layer is composed of the following components:

  - **Reasoner**: When the context model is updated, the reasoning component computes a derived variability model that only contains the mandatory features and a selection of variable features, adapted to the current context. This component is initialized with a variability model and a reasoning model.

- **Weaver**: This component receives a pruned variability model from the reasoning component. For all the features of the variability model, the weaver composes the corresponding aspect models to produce a target configuration. This configuration is then checked before it is submitted (if valid) to the causal link (proxy layer).

- **Checker**: This component checks that the produced configuration is consistent [105]. It checks all the invariants that should be enforced by the business system.

- **Proxy layer**: This layer is responsible for bridging the gap between design-time models and the runtime. It is composed of:

  - **Monitoring**: This component observes runtime events generated by probes integrated into the runtime in order to create and update a context model. We do not impose any particular monitoring technique. For example, in Section 5.3, we propose to use a multi-staged monitoring approach combined with Complex Event Processing techniques.

  - **Causal Link**: This component receives an architectural model to configure and reconfigure the business architecture using the services offered by the execution platform, as described in Section 5.4.

- **Business layer**: This layer basically contains all the business components that are managed by the reference architecture.

The interactions between these components are specified by the sequence diagrams shown in Figure 6.2

The context model is updated when relevant changes appears in the execution context of the running system (*e.g.*, CPU load, free memory or bandwidth). This context model is really an interface between the reference architecture and the execution context. This model is kept in a shared memory. Indeed, since it is frequently updated, it does not make sense to make this model persistent. Components that require this model (*e.g.* the Reasoner) simply register to the notification mechanism provided by EMF.

The architectural model is updated when the running system evolves *i.e.*, when components and bindings are added or removed. It is important to note that this architectural model is not directly manipulated in order to adapt the running system. On the contrary, we produce **another** architectural model (configuration) when the system should adapt [105]. This makes it possible to validate the new configuration before the actual adaptation, and fully automate the reconfiguration process by reasoning of the former configuration and the new configuration [105]. This prevents the architect from writing low level and error-prone reconfiguration scripts. In the case the new configuration is not valid, we simply discard this configuration. Indeed, since the running system has not been adapted yet, it is not necessary to perform a rollback.
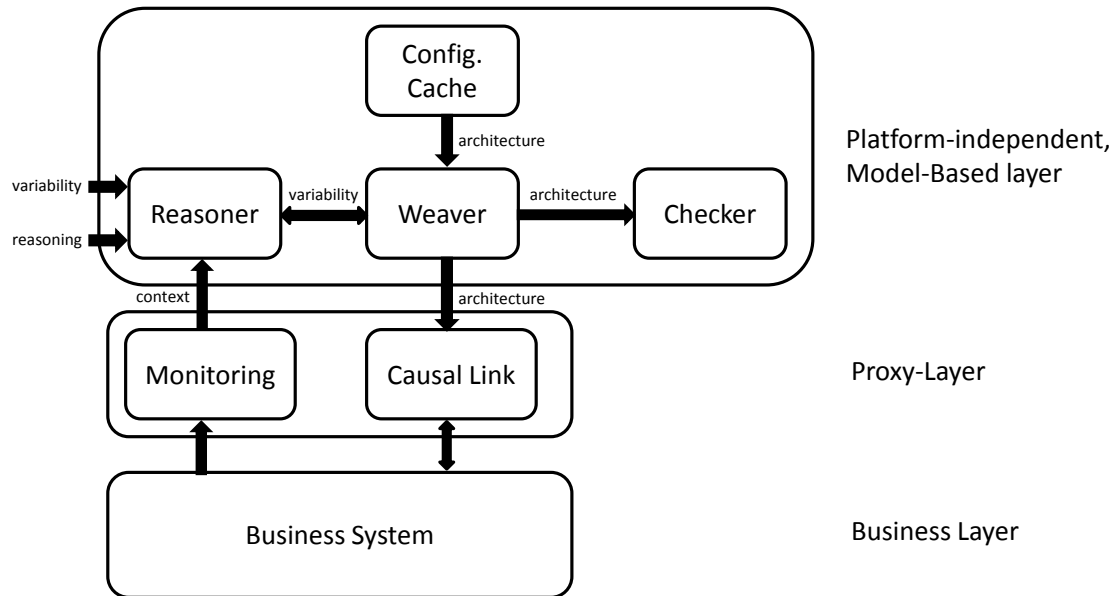
Figure 6.1: Runtime Architecture to Support DSPLs

The platform-independent, model-based layer, which is used at runtime, can also be employed at design-time, since it only consumes and produces design models. This way, it is possible to perform early simulations [50] or tests before actually deploying the adaptive system, using the same infrastructure than the one that will be actually used at runtime. The basic idea is to mock the whole layer (or individual components) with stub components that produce and consume models. These models are either provided by the designer (*e.g.*, context models corresponding to critical situations) or can be automatically generated [113, 141]. We also automatically generate a simple graphical user interface (see Figure 6.3 and Appendix A.5 for the code generator) so that it is possible to dynamically simulate the context instead of creating context models by hand.

6.1   Communication Schemes

The reference architecture is aligned with the Monitor-Analysis-Plan-Execute (MAPE) loop [81]. The main tasks associated with the MAPE loop are encapsulated as components. These components mainly exchange models, which conform to the metamodels we have presented in Section 3, according to different schemes:

- by URI: The producer component serializes the model and pass the URI to the con-
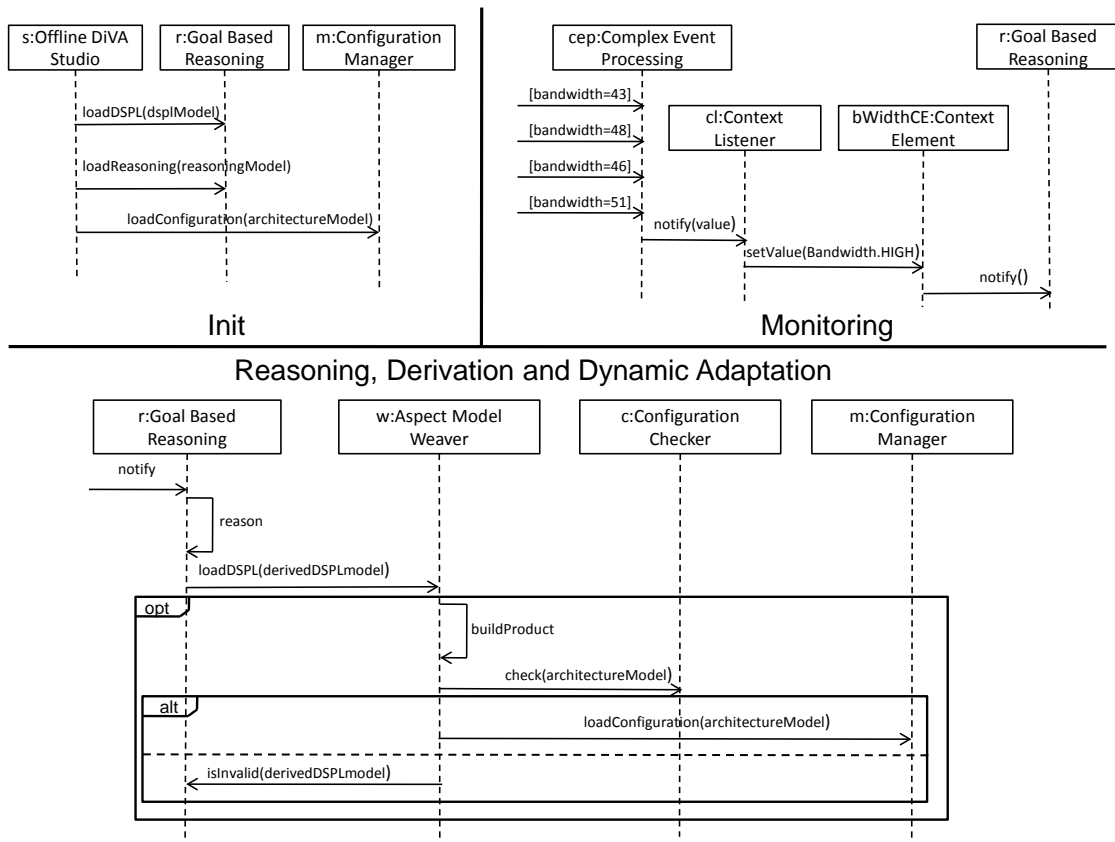
Figure 6.2: Interactions between the components

sumer component, so that it will be able to load this model. We use the serialization
capabilities provided by EMF to easily load and save models from/to an URI. Serial-
ized models can efficiently be zipped (because XMI files contains recurrent patterns)
and quickly transmitted via network links. This is similar to a call by value.

- by Resource: The producer component directly pass the Resource (graph of objects
  in memory) containing the model to the consumer component, which can directly
  manipulate the model. This is similar to a call by reference.

- by Notification: The producer manipulates the model (in memory), and the con-
  sumer is notified from the changes by the EMF notification mechanism. In this case,
  there is no direct exchange between the producer and the consumer: the producer
  works on a resource (model) that is observed by the consumer. This is a shared
  memory between the consumer and the producer.
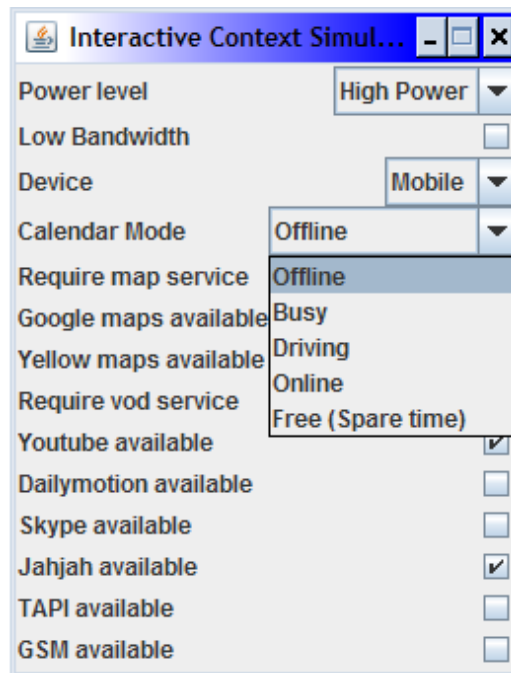
Figure 6.3: Generated GUI to simulate the context

The *URI* and *Resource* schemes are very similar in practice. Indeed, to actually manipulate a serialized model, a component has to load it into a resource. Typically, if a component offers a service that manipulates a model using a resource or (xor) an URI, it can easily be refactored so that it can leverage both the URI and the resource, as illustrated in the script below:

```
1    //Causally connected models
2    private art.System system; //the reflection model
3    private art.System updateModel;//a target model
4
5    public void reconfigureByModelURI(String modelURI) {
6        loadUpdateModel(modelURI);
7        reconfigure();
8    }
9
10   public void reconfigureInMemory(art.System targetSystem){
11       updateModel = targetSystem;
12       reconfigure();
13   }
14
15   /**
```

```
16      * Actually reconfigure the running system using updateModel
17      */
18     private void reconfigure(){
19       ...
20     }
```

6.2   Different Configurations for the Reference Architecture

This section presents different possible configurations for the reference architecture supporting the approach presented in this thesis. Following the idea of the FraSCAti implementation [140] of the SCA standard, this reference architecture is itself developed as a component-based application. It thus makes it possible to finely configure (and even to dynamically reconfigure, as discussed in Section 9.2) the adaptation loop according to the need.

6.2.1   The Case of Small Adaptive Systems

In the case of small adaptive systems, where the total number of configurations remains manageable by hand, it is possible to generate and validate all the possible configurations at design-time and define the adaptation logic using simple ECA rules. At runtime, the adaptation loop is thus configured as illustrated in Figure 6.4. In particular the *Weaver* component simply chooses already woven configuration from a cache and does not perform weaving at runtime.
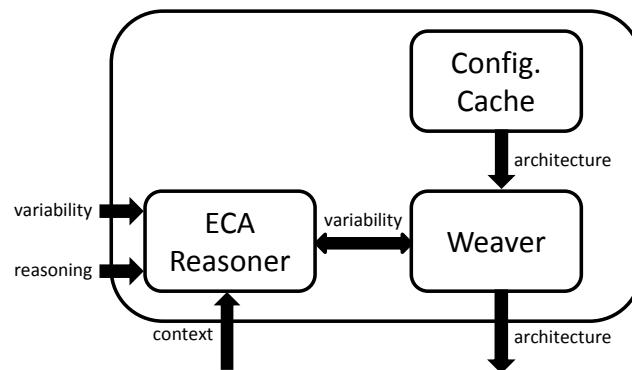
Figure 6.4: Runtime Architecture for Small Adaptive Systems

This particular configuration of the adaptation loop is rather efficient, since most of the tasks (weaving, validation) are performed at design-time. This is rather similar to what

Bencomo *et al.* [15, 12, 13, 14, 17] or Zhang *et al.* [165] propose. However, this kind of configuration of the adaptation loop is not adapted to complex adaptive systems that cannot be fully validated at design-time due to the explosion of the number of configurations.

6.2.2 ECA, Goals and Online generation/validation of Configurations to Tame Complex Adaptive Systems

In the case of complex adaptive systems, such as the ones addressed in the DiVA project, it is not possible to enumerate and validate all the possible configurations of a DAS at design-time. At runtime, the adaptation loop is thus configured as illustrated in Figure 6.5. This loop use an ECA-like reasoner and a goal optimizer in combination. Since all the reasoner components consume and produce variability model, it is possible to chain several reasoners in order to derive the variability model in several steps. Depending on the context, the two reasoners compute a derived variability model that is transformed into an architectural model by the weaver. If a corresponding configuration already exists in cache, then it is directly reused. Otherwise, all the aspect models corresponding to the derived variability model are actually woven. Since all the configurations cannot be validated at design-time, all the woven configuration are checked at runtime. In a configuration is valid, it is submitted to the causal link component, to drive the dynamic reconfiguration. Otherwise, the weaver informs the reasoner that this particular configuration is invalid, which updates the constraints of the variability model in order not to derive this configuration anymore.
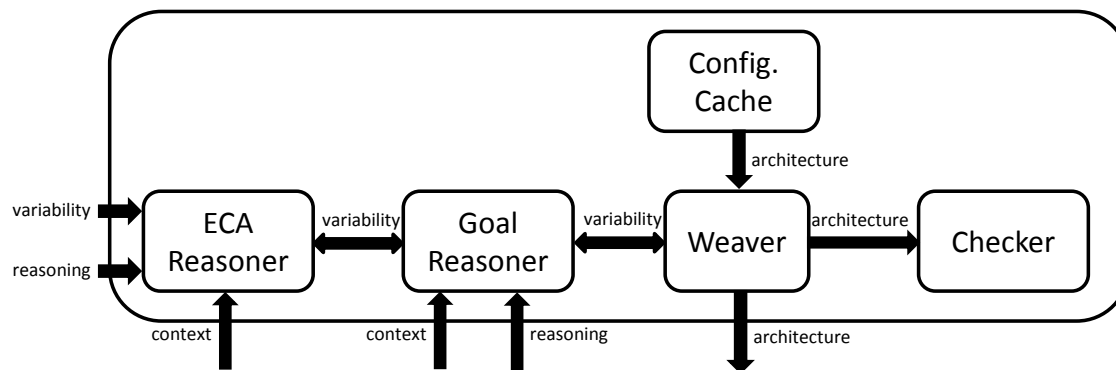
Figure 6.5: Runtime Architecture for Complex Adaptive Systems

This configuration of the adaptation loop combines several reasoning paradigms and is able to weave and validate configurations at runtime. This configuration is thus well-adapted to complex adaptive systems that cannot be designed in extension and that can-

not be fully validated at design-time.

6.3 Discussion

This reference architecture supports the different contributions presented in the previous chapters of this part. It can be used at design-time (for simulation purpose) or at runtime to support the dynamic adaptation process. Indeed, it mainly manipulates design models, even at runtime. This architecture is very modular and it can be configured into different ways depending on the complexity of the DAS it has to manage. This architecture has been employed by different users to realize various case studies, as described in the next chapter.

Chapter 7

Validation: Application to 2 Case Studies

Contents

7.1 Objectives

In this Chapter, we apply our approach to two different case studies (in addition to the running example (D-CRM) that was presented in the previous chapter). The first case study is conducted by an industrial partner of the DiVA European project. The second one is currently incubated in the academic world and may be industrialized in a near future. It is important to note that these 2 case studies (+ the D-CRM presented along the previous chapters) are not conducted by people involved in the development of the tools supporting the work described in this thesis. This is an important remark: this means that the tools developed in the context of this thesis (and more largely, the tools developed within the DiVA project) are actually usable by industrial and academic people with no insight of these tools.

We illustrate the design and aspect weaving steps on the first case study, since we only have access to the design artifacts of this case study *i.e.*, we cannot run the system ourself to perform experiments. The model-driven dynamic reconfiguration will thus be validated on the second case study developed within the INRIA Triskell Team.

The objectives of this validation are to evaluate:

- The modeling step (variability, context, reasoning). In particular, we will evaluate if this modeling step is able to tame the explosion of the number of artifacts a designer has to specify to describe and engineer a DAS.

- The weaving of aspect models. In particular, we will evaluate the expressiveness of SmartAdapters (is it able to automatically produce woven models that make sense for a designer), its performances and its scalability (is it able to weave a sequence of aspects in a reasonable time).

- The model-driven dynamic adaptation process. In particular, we will evaluate its performances.

7.2   Validation on a Crisis Management System

The Crisis Management system is a critical system currently developed by Thales[1] in the context of the DiVA project. The objectives of the Crisis Management system is to handle crisis situations in an airport (*e.g.*, crash of an airplane on a runway). This application runs on ServiceMix[2] an open-source ESB (Enterprise Service Bus) by the Apache foundation, based on the Felix/Karaf OSGi runtime[3].

7.2.1   Design of the Crisis Management System

This sub-section describes the different design models of the crisis management system.

Variability

Figure 7.1 shows the variability of the crisis management system. This system should for example be able to connect to several external organizations, such as Hospitals, Fire department, etc using different notification and planning strategies.

This variability model (5 Dimensions, 20 variants) leads to 30720 possible configurations[4], and 943,687,680 possible transitions[5] among these configurations.

---

[1] http://www.thalesgroup.com
[2] http://servicemix.apache.org
[3] http://felix.apache.org/site/apache-felix-karaf.html
[4] $2^5*3*3*2^5*5 = 30,720$
[5] 30,720*30,719 = 943,687,680

	Name	ID	Lower	Upper
⊟ Dimension	Organization	ORG	0	-1
Variant	Hospital	HOSP	-	-
Variant	FireDepartement	FIREDEP	-	-
Variant	Police Office	POLICE	-	-
Variant	Minestry	MINESTRY	-	-
Variant	TVChannel	TV	-	-
⊟ Dimension	Notification	ORGRK	1	1
Variant	DefaultNotificationStrategy	DEFAULTRK	-	-
Variant	EmergencyNotificationStrategy	EMERGRK	-	-
⊟ Dimension	SensorDeploymentStrategy	DEPLOYMENTSTRATEGY	1	1
Variant	FireStrategy	FIRESTRATEGY	-	-
Variant	AttackStrategy	ATTACKSTRATEGY	-	-
Variant	EnergySaving	ENERGYSAVE	-	-
⊟ Dimension	Planner	PLANNER	0	-1
Variant	FirePlanner	FIREPLANNER	-	-
Variant	MedicalPlanner	MEDICPLANNER	-	-
Variant	NewsPlanner	NEWSPLANNER	-	-
Variant	TacticalPicturePlanner	TACPICPLANNER	-	-
Variant	TerrorismPlanner	TERRPLANNER	-	-
⊟ Dimension	Logging	LOG	1	1
Variant	Normal	NORMLOG	-	-
Variant	Critical	CRITLOG	-	-
Variant	Warning	WARLOG	-	-
Variant	Push	PUSHLOG	-	-
Variant	Push Broadcast	BROALOG	-	-

Figure 7.1: Variability of the Crisis Management System

Environment

Figure 7.2 shows the specification of the environment for the crisis management system. The important aspects of the context are the alert level, the type of crisis (fire, weather, etc), if there are damages or victims, etc.

Reasoning

Figure 7.3 illustrates some basic rules that filters the variability depending on the context. The connection to external organizations is very specific to the context: each *variant* of the organization *dimension* is associated with an *available* and a *require* rule. In other words, there is an equivalence between some context fragments and the choice of the organizations. Since the crisis management system is a critical system, the designers intensively rely on ECA-like rules, so that they can have control on the choice of the variants of the system. 20 ECA-like rules[6] are sufficient to describe obvious adaptation rules.

However, the designers also use higher level goals to fully design the adaptation logic of their system, by defining impacts and priority rules. Figure 7.4 illustrates the impact of

---

[6]if we consider that an *available* rule and a *require* rule with the same parameters define a single *equivalent* rule.

	Name	ID	Values
⊞ ≣ Enum	AlertLevel	ALERTLV	{NORM, HIGH, LOW, MED, VHIGH}
B Boolean	FailedAuthentication	FAUTH	-
⊞ ≣ Enum	Crisis Location	CRLOC	{RUN1, RUN2, TERM1, TERM2, INN, PARK1, PARK2}
B Boolean	PhysicalDamages	DAMAGES	-
⊞ ≣ Enum	UserLocation	USERLOC	{INCR, OUTCR}
B Boolean	ExistenceFingerPrintReader	FINGERREAD	-
⊞ ≣ Enum	RescuerRole	RESROLE	{PHYSICAL, MEDICAL, OBSERVER}
B Boolean	RequiredInformationExtPeople	INFO	-
B Boolean	Victims	VICTIMS	-
B Boolean	RequiredEscalation	ESC	-
B Boolean	SuspiciousBehavior	SUSPBEHAV	-
⊞ ≣ Enum	CrisisType	CRTYPE	{FIRE, ATTACK, CHEMSPILL, EXPL, WEATHER, CATAS, NULL}
⊞ ≣ Enum	InnerFourCourtUser	COURTUSER	{PERS, TEMP}
B Boolean	Default	DEFAULT	-

Figure 7.2: Environment of the Crisis Management System

the variants on QoS properties.

Figure 7.5 shows the different priority rules responsible for defining the goals of the system according to the context. 8 rules are sufficient to describe the goals of the system.

7.2.2   Some Aspect Models of the Crisis Management System

In this sub-section, we illustrate some of the aspect models that refine the variants of the variability model. The log aspects are very similar to the aspect presented in Section 4.4. The fire planner aspect, illustrated in Figure 7.6, is also quite similar to the log aspect. It basically consists in linking the 3 ports provided by the fire planner component to any component that require these 3 ports. The fire planner component is unique *i.e.*, it will be introduced only once, even in the case of multiple join points. In the pointcut, all the required ports are associated to a single component. This means that a component of the base model can match this component only if it requires (at least) all these 3 ports.

A more interesting aspect is the fire department aspect, which is responsible for connecting the system with an external organization (fire department) using a SOAP channel and a SMTP channel. The advice model is composed of 3 components, with some bindings among them. The two channel components are connected to any component that can push information the channel can manipulate. They are also connected to another component that can receive feedback from the two channel components.

Figure 7.8 illustrates one possible configuration of the crisis management system. The base model is initially composed of 2 components: *ThinkingTool* and *ContextManager*. Then, two planners (*FirePlanner* and *TacticalPicturePlanner*) are woven. Finally, the aspect associated with the fire department is woven. As illustrated in Figure 7.7, the advice is composed of three unique components: *FireDeptFilter*, *FireSOAPChannel* and *FireSMT-PChannel*, as well as some bindings that connect these components. The pointcut is com-

	Name	ID	available	required
⊟ Dimension	Organization	ORG	-	-
Variant	Hospital	HOSP	VICTIMS	VICTIMS
Variant	FireDepartement	FIREDEP	CRTYPE=FIRE or CRTYPE=EXPL or DAMAGES or …	CRTYPE=FIRE or DAMAGES or CRTYPE=EXPL or CRTYPE…
Variant	Police Office	POLICE	CRTYPE=ATTACK or SUSPBEHAV	CRTYPE=ATTACK or SUSPBEHAV
Variant	Minestry	MINESTRY	ESC	ESC
Variant	TVChannel	TV	INFO	INFO
⊟ Dimension	Notification	ORGRK	-	-
Variant	DefaultNotifica…	DEFAULTRK		not (ALERTLV=HIGH or ALERTLV=VHIGH)
Variant	EmergencyNoti…	EMERGRK		ALERTLV=HIGH or ALERTLV=VHIGH
⊟ Dimension	SensorDeploym…	DEPLOYMENT…	-	-
Variant	FireStrategy	FIRESTRATEGY		CRTYPE=FIRE
Variant	AttackStrategy	ATTACKSTRA…		CRTYPE=ATTACK
Variant	EnergySaving	ENERGYSAVE		CRTYPE=NULL
⊟ Dimension	Planner	PLANNER	-	-
Variant	FirePlanner	FIREPLANNER	CRTYPE=FIRE or CRTYPE=EXPL or CRTYPE=CA…	CRTYPE=FIRE or CRTYPE=EXPL or CRTYPE=EXPL or CRT…
Variant	MedicalPlanner	MEDICPLANNER	VICTIMS	VICTIMS
Variant	NewsPlanner	NEWSPLANNER	INFO and (CRTYPE=EXPL or CRTYPE=CATAS or …	INFO and (CRTYPE=EXPL or CRTYPE=CATAS or CRTYPE…
Variant	TacticalPicture…	TACPICPLAN…	CRTYPE=ATTACK or CRTYPE=CATAS or ALERTL…	CRTYPE=ATTACK or CRTYPE=CATAS or ALERTLV=HIGH …
Variant	TerrorismPlanner	TERRPLANNER	CRTYPE=ATTACK	CRTYPE=ATTACK
⊟ Dimension	Logging	LOG	-	-
Variant	Normal	NORMLOG		ALERTLV=NORM
Variant	Critical	CRITLOG		ALERTLV=HIGH
Variant	Warning	WARLOG		ALERTLV=LOW
Variant	Push	PUSHLOG		ALERTLV=MED
Variant	Push Broadcast	BROALOG		ALERTLV=VHIGH

Figure 7.3: ECA-like rules of the Crisis Management System

posed of two components, leading to two overlapping join points:

1. *Any → TacticalPicturePlanner*; *Another → ThinkingTool*

2. *Any → FirePlanner*; *Another → ThinkingTool*

These two overlapping join points are seamlessly handled by the default scoping strategy associated with the bindings. Without this strategy, the bindings between the **Channel* components and the *ThinkingTool* component would be duplicated, leading to an erroneous configuration.

7.2.3 Comparative Study of SmartAdapters V1 and V2

In this subsection we compare the initial version of SmartAdapters, with the newly developed one, on a real-life scenario consisting of 5 configurations (see Appendix B.1). The new version of SmartAdapters totally outperforms the old one and is able to weave all the configurations in less than one second. The simplest configuration (Before Crisis: 3 aspects and 4 join points in total) is woven in less than 300 ms (163.9 times faster than V1), while the most complex configuration (Fire Spreads: 9 aspects and 32 join points in total) is woven in less than 900 ms (649.6 times faster than V1).

An important result illustrated by Figure 7.9 is that the time needed to weave an aspect is almost constant (100 ms), whatever the number of aspects. In other words, the size of the base model seems to have a very limited impact on the time needed to weave an aspect. This new version of SmartAdapters is faster and much more scalable than

	Security	RescueCapacity	RescueResponseTime	Traceability	InformationQuality	Confidentiality	NetworkLatency
Organization (ORG)	false	false	false	true	false	true	false
Hospital (HOSP)	-	-	-	Very Low	-	Low	-
FireDepartement (FIREDEP)	-	-	-	Very Low	-	Medium	-
Police Office (POLICE)	-	-	-	High	-	High	-
Minestry (MINESTRY)	-	-	-	Very Low	-	Very High	-
TVChannel (TV)	-	-	-	Very Low	-	Very Low	-
Notification (ORGRK)	false	true	true	false	false	true	true
DefaultNotificationStrategy (DEFAU	-	Low	High	-	-	High	Low
EmergencyNotificationStrategy (EMI	-	High	Low	-	-	Low	High
SensorDeploymentStrategy (DEPLOYME	false	false	false	false	true	false	false
FireStrategy (FIRESTRATEGY)	-	-	-	-	Very High	-	-
AttackStrategy (ATTACKSTRATEGY)	-	-	-	-	High	-	-
EnergySaving (ENERGYSAVE)	-	-	-	-	Low	-	-
Planner (PLANNER)	false	false	false	false	true	false	false
FirePlanner (FIREPLANNER)	-	-	-	-	Medium	-	-
MedicalPlanner (MEDICPLANNER)	-	-	-	-	Medium	-	-
NewsPlanner (NEWSPLANNER)	-	-	-	-	Medium	-	-
TacticalPicturePlanner (TACPICPLAN	-	-	-	-	High	-	-
TerrorismPlanner (TERRPLANNER)	-	-	-	-	High	-	-
Logging (LOG)	true	false	false	true	false	false	true
Normal (NORMLOG)	Very Low	-	-	Very Low	-	-	Very High
Critical (CRITLOG)	Medium	-	-	Medium	-	-	Medium
Warning (WARLOG)	Low	-	-	Low	-	-	High
Push (PUSHLOG)	High	-	-	High	-	-	Low
Push Broadcast (BROALOG)	Very High	-	-	Very High	-	-	Very Low

Figure 7.4: Impacts of the variants of the Crisis Management System on QoS properties

	Name	ID	context	Security	Traceability	InformationQuality	Confidentiality	NetworkLatency
Rule	High alert	hIGHalt	ALERTLV=HIGH	High	High	High	High	Low
Rule	Explosion	EXPLR	CRTYPE=EXPL	High	High	High	Medium	Low
Rule	Low alert	LowAlt	ALERTLV=NORM	Medium	Low	Low	Very Low	High
Rule	User in control room	INCRR	USERLOC=INCR	Low	-	-	Low	-
Rule	User out control room	OUTCRR	USERLOC=OUTCR	High	-	-	High	-
Rule	SUSPICIOUS BEHAVIOR	SUSPBEHAVR	SUSPBEHAV	Very High	High	High	Very High	Medium
Rule	Attack	ATTACKR	CRTYPE=ATTACK	High	High	High	Very High	Low
Rule	Victims	VICTIMSRULE	VICTIMS	-	High	High	-	Low

Figure 7.5: Goals of the Crisis Management System

the previous one in which the average time to weave an aspect is well correlated to the number of aspects to weave and the number of join point to detect.
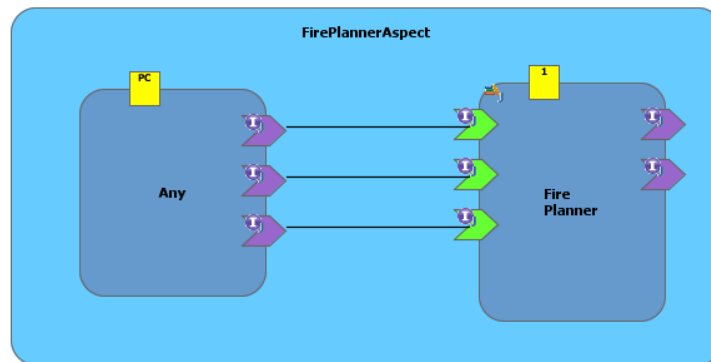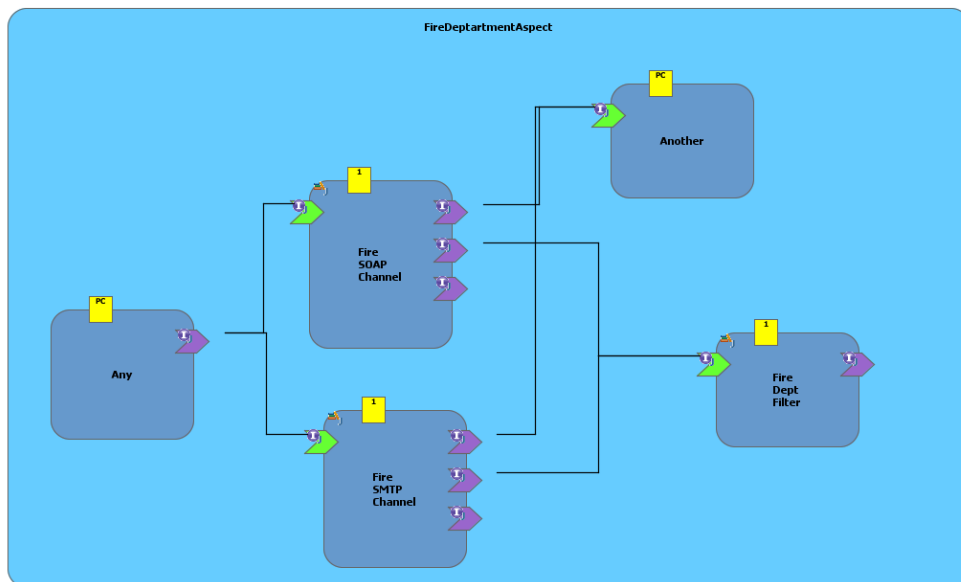
Figure 7.6: The Fire Planner aspect

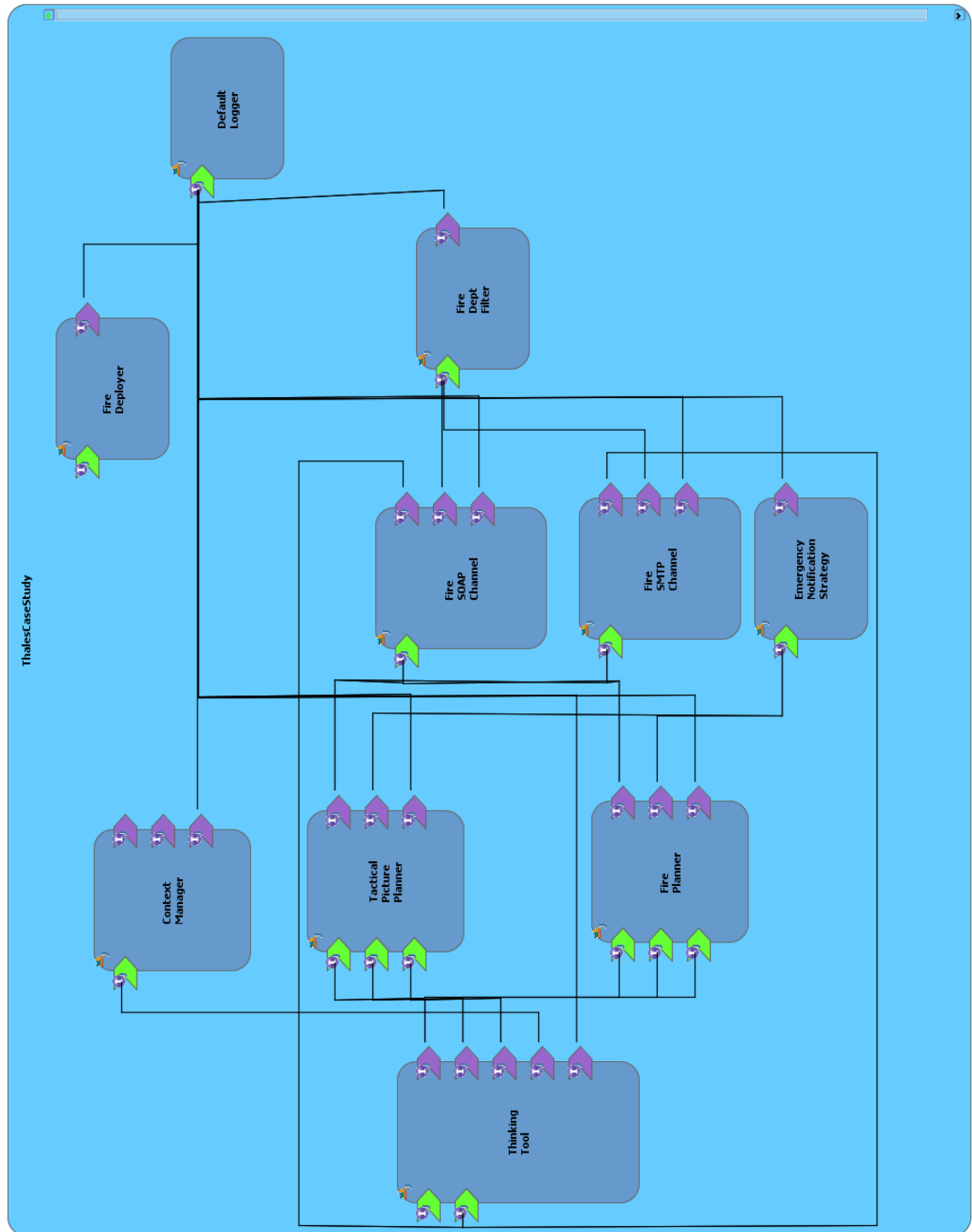Figure 7.7: The Fire Department Aspect

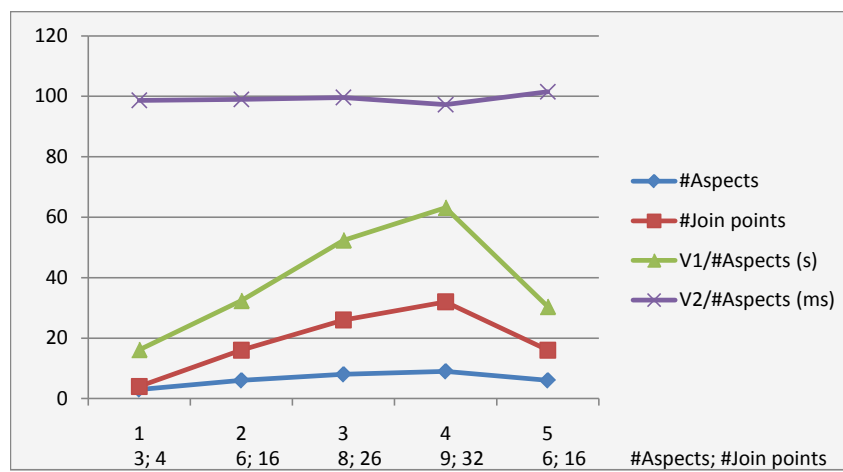Figure 7.8: A typical configuration of the Crisis Management System

Figure 7.9: Performances of SmartAdapters V1 and V2 for the Crisis Management system

7.3   Validation on EnTiMid

EnTiMid [105, 116, 115][7] is a middleware for Home-Automation systems incubated by the INRIA research center. The goal of EnTiMid is to assist elderly and/or disabled people in their everyday life by automating some tasks in their homes (Ambient Assisted Living or AAL). This application runs on Felix/Karaf on a MSI Wind[8] with a touch screen.

7.3.1   Scenario

The scenario we propose to illustrate is the following (see Figure 7.10):

1. **Initial Deployment**: the system is deployed during the day, and configured to meet the requirement of the elderly person living in the intelligent house. In this configuration, the elderly person can send emergency messages via SMS to a control center by using a simple device with only one button. While the SMS is sent, the person is notified via a device equipped with text-to-speech capabilities. In this configuration, the system is composed of 18 components (OSGi bundles or simple components in memory) and 9 bindings among these components.

2. **Next two days**:

    (a) **Night**: when the night falls, or more precisely when the light detector sends values lower than a given threshold, the application is reconfigured. The emergency feature is still active. In addition, some lights of the flat are switched on in the case of an emergency, so that the flat will be sufficiently lit when the medical staff arrives. In the case of a false positive emergency message (*e.g.* when the elderly person can answer the phone), the lights can be remotely switched off using the XMPP protocol. In this configuration, the system is composed of 20 components and 18 bindings among these components.

    (b) **Day**: when the day arises, the system is reconfigured into the initial configuration (step 1). Then, step 2 iterates one more time.

    (c) **Day + nurse visiting**: a nurse arrives to check the status of the patient

    (d) **Day (after the nurse has left)**

This scenario has been publicly demonstrated at AOSD'10 [107].

7.3.2   Results

The results of this experiments are illustrated in Figure 7.11 (see Appendix B.2 for the precise times). During the initial configuration, all the components needs to be deployed.

---

[7]"en ti" means "at home" in brezhoneg, the language (not so) spoken in Britanny.
[8]CPU: Intel Atom 230@1.6GHz, RAM: 1Gb and OS: Ubuntu 9.10 (Linux kernel: 2.6.31-17-generic)

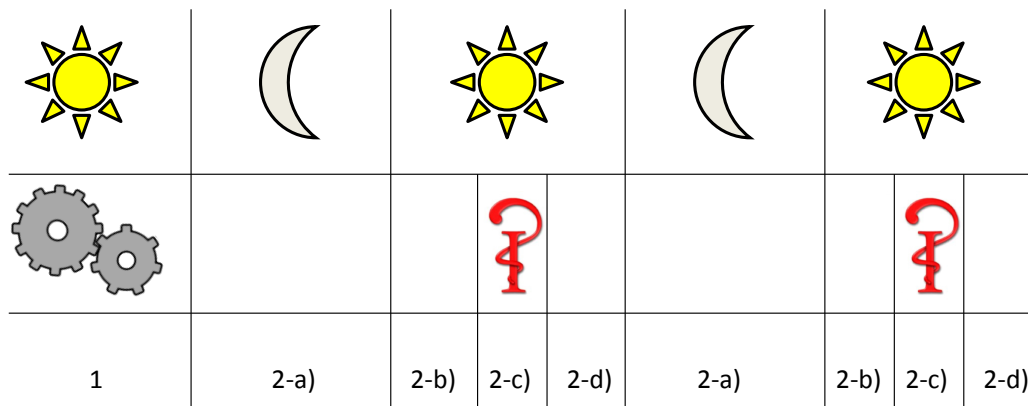| 1 | 2-a) | 2-b) | 2-c) | 2-d) | 2-a) | 2-b) | 2-c) | 2-d) |

Figure 7.10: A typical AAL scenario

In particular, all the components that need to be wrapped into OSGi bundles have to be compiled and packaged at runtime. This explains the rather long reconfiguration time of step *1*: 5.75 seconds (not shown in the graphic). During the first reconfiguration (day $\rightarrow$ night, step *2-a)*) all the already deployed components are reused. In addition, the component responsible for the lights has to be compiled, packaged and properly deployed and connected to other components. This is realized in less than 400 ms. The next 3 reconfigurations (steps *2-b)*, *2-c)* and *2-d)*) are much more fast. Indeed, we maintain a cache of pre-generated components (jar files) that are directly reused, with no need to compile and package them. The next 4 reconfigurations (corresponding to the second iteration of step *2*) are also reasonably fast.

When the cache of components is initialized (after the first iteration of step *2-a)*) the average reconfiguration time is about 60 ms.

For each reconfiguration, we first perform a model comparison. This model comparison takes an almost constant time of 430 ms to compare models with about 20 components. This comparison step is executed before the actual reconfiguration to plan the reconfiguration (*i.e.* instantiate the reconfiguration commands). It thus delays the actual reconfiguration of the system but does not impact the (un)availability of some components during dynamic reconfiguration.

7.3.3 Discussion

Our Model-Driven dynamic adaptation process is obviously less efficient (*w.r.t.* to the time needed to actually reconfigure an application) than hard-coded reconfiguration scripts, for several reasons:

1. We systematically perform a global model comparison between the source and the
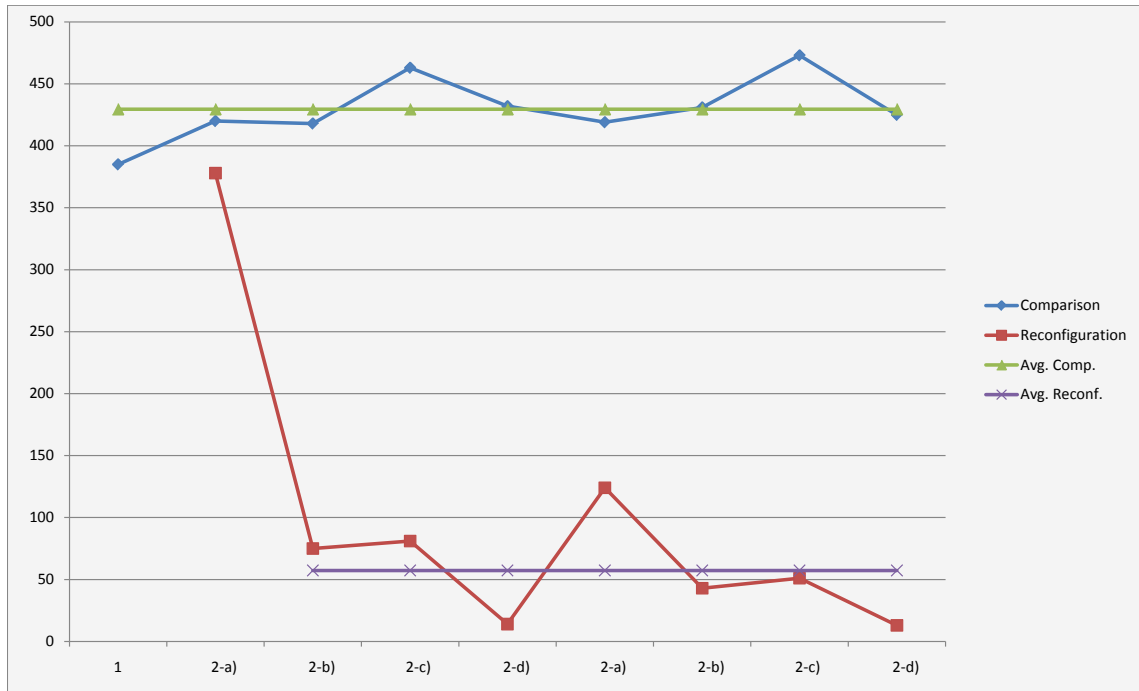
Figure 7.11: Time (in ms) spent in Configuration Comparison and Actual Reconfiguration after the initial deployment

target configuration to determine all the actions we need to reconfigure the system.

2. Some components (embedded into OSGi bundles) are compiled, packaged and deployed at runtime.

However, these two drawbacks have significant advantages, and can easily be minimized:

1. The model comparison prevents architect from writing low-level and error-prone reconfiguration scripts. Instead, the system automatically computes a safe reconfiguration script that takes care of the life-cycle of components. It is important to note that this model comparison does not increase the actual dynamic reconfiguration time, it simply delays the reconfiguration. In other words, it does not impact the availability of services. Moreover, this model comparison can be performed by a third-party system (more powerful than a MSI Wind), which would return a list of reconfiguration actions. Indeed, each configuration of EnTiMid is serialized in about 30 Kb (only a few Kb if zipped) so that it can be quickly transmitted on a

network. Finally, as we discussed in Section 4.7, it could be possible to avoid this comparison, by making the causal link specific to a given aspect model weaver (*e.g.* SmartAdapters), which would instantiate the reconfiguration commands during the weaving.

2. As we have seen in our experiment, the fact that an OSGi component is already available in cache significantly improves the performances of the reconfiguration process. The cache of OSGi components can easily be initialized by a component that iterates on a set of configurations and periodically (*e.g.* every 5 or 10 seconds) reconfigures the application with the next configuration. This way, it is possible to initialize the cache in a few minutes after the installation of EnTiMid. Moreover, the compiling and packaging of a component that would not be present in cache can also be externalized to a third system that would simply return the jar file of the required component.

Part III

Conclusion and Perspectives

Chapter 8

Conclusion

This thesis presented a Model-Driven and Aspect-Oriented approach to tame the complexity of adaptive systems [105, 147]. Following the recent trend of *models@runtime* [114, 18], this approach leverages models both at design-time and at runtime to support the dynamic variability of complex adaptive systems. We use various Domain-Specific Modeling Languages to capture the different facets of an adaptive system:

1. **Variability**: describes the different features of the system, and their natures (options, alternatives, etc)

2. **Environment/Context**: describes the relevant aspects of the context we want to monitor (environment), as well as the current context.

3. **Reasoning**: describes when the system should adapt. It basically consists in defining which features (from the variability model) to select, depending on the current context.

4. **Architecture**: describes the configuration of the running system in terms of architectural concepts.

The two keys of this thesis to tame dynamically adaptive systems are *abstraction* and *modularization*. Models conforming to these metamodels are available both at design-time (prior to the initial deployment) and runtime; offline and online (see Definitions in Section 5.5). This totally blurs the traditional boundary between design-time and runtime. An important point is that models embedded at runtime are really mirrors of what really happens in the running system. We have shown that it is possible to work on copies of these models (which are independent of the running system) and to re-synchronize these copies with the reality to actually adapt the running system. In other words, our approach makes it possible to perform offline activities such as continuous design or prediction,

while the system is running, but independently from it. The results of these offline activities can then feed online activities, which can reuse already computed results to improve the performances of the dynamic adaptation.

To prevent the designers from specifying all the possible configurations of an adaptive system, we leverage Aspect-Oriented Modeling techniques and especially the SmartAdapters [18] approach. We refine each feature of the adaptive system into an aspect model (a fragment of architecture). This way, it is possible to automatically derive a huge set of architectures by combining a relatively small set of aspects. Current AOM weavers are supposed to be used at design-time, where performance issues are not critical. However, this becomes an important issue at runtime, especially when the weaving should be performed online. Moreover, current AOM weavers do not provide enough expressiveness to be able to weave cross-cutting aspect models in a consistent and meaningful way. We thus extend the SmartAdapters weavers we developed with the original idea of advice sharing that let designers specify how the advice should be managed when the pointcut matches several join points. We also significantly improved the performances of this weaver, especially the performance of the join point identification. In our implementation, aspect models are now compiled into scripts that can directly be applied at runtime, with no need to run an interpreter or to map aspects to another formalism.

Let us conclude by positioning our contributions *w.r.t.* the questions asked in the introduction Chapter (Section 1.2):

1. *How to abstract the actual execution context into a high-level model, which offers a solid basis for reasoning and decision making, and hides irrelevant details?*
   We adopt a step-wise approach to progressively abstract the real execution context (a flow of data provided by probes) into a model of the context. In particular, we extend WildCAT so that it is able to monitor models. This way, we can build intermediate context models instead of bridging the gap in one step with very complex queries. We do not impose any specific techniques to produce these intermediate context models. The most abstract context model is a qualitative context model, which allows designers to define the adaptation logic in a simplified manner because it totally hides technical details (how the monitoring is actually performed).

2. *How to efficiently reason on the current context in order to find a well-suited set of features, with no need to enumerate all the possible contexts or to enumerate numerous rules?*
   The actual adaptation logic is fully defined on the most abstract context model, modeled at design-time. We propose two kinds of formalisms to define the adaptation logic: (*i*) simple rules that directly relate context fragments to the activation/deactivation of features (rather similar to human *reflex*), and (*ii*) high level goals that the system should optimize based on the impact of the features on QoS properties (rather similar to human *thinking*). Each reasoning component has well defined interfaces. It takes as input a variability model, the current model of the context and a model describing the adaptation logic (simple rules, goals, etc). It produces a

pruned variability model where some variation points have been fixed. It is thus possible to chain several reasoning components to progressively derive a variability model that corresponds to a single product. The simple rules are rather intuitive to define and easy to understand in isolation and they can efficiently prune the variability model. However, it rapidly becomes difficult to specify the whole adaptation logic using these simple rules because it would lead to a large set of rules they can possibly interact. Goals are particularly well suited to describe QoS-based adaptation logic with a small set of optimization rules. However, the runtime optimization of goals could be rather costly. The combination of several reasoning formalisms offers a good trade-off between performances, comprehension and maintainability.

3. *How to avoid the adaptive system to continuously oscillates when the context slightly oscillates around critical thresholds?*
The trade-off between stability and reactivity is very difficult to achieve and really depends on the type of adaptive system. This is why we do not impose one particular technique. It is for example possible to use Complex-Event-Processing techniques or hysteresis cycles (to improve the stability), simple thresholds (to improve the reactivity), or a combination of different techniques.

4. *How to actually build the configuration (architecture) corresponding to a set of features, with no need to fully specify all the possible configuration beforehand, while preserving a high degree of validation?*
Each feature of the system is refined by an aspect model, which is a modularization unit that embeds all the information needed so that it can easily be composed with a base model (which contains all the mandatory elements). Since we are interested in component-based reconfiguration, an aspect model is a fragment of architecture. In particular, we rely on the SmartAdapters weaver we developed at the very beginning of this thesis. We extend this weaver with the original notion of advice sharing, which allows designer to describe expressive architectural aspects that yield to meaningful woven models. SmartAdapters combines declarative and imperative techniques to model the aspect models. This prevents designers from writing low-level scripts needed to integrate their aspects. Our latest implementation of SmartAdapters leverages model-driven and generative techniques to significantly improve the performances of the aspect model weaving. The validation chapter shows that each aspect model can be woven in an almost constant time of 100 ms in realistic case studies. Designers never specify the whole possible set of configurations. Instead, this huge set is described in intention by the variability model and each configuration is automatically built when needed. When a new woven model is produced, the running system is not directly adapted. The reflection model (which abstracts the running system) is never directly manipulated by the adaptation chain. Instead, the dynamic adaptation chain produces independent woven models that represent the configuration that the running system should mi-

grate to. Before actually reconfiguring the running system, this produced model is first validated to ensure its consistency. This way, the running system will never switch towards an erroneous configuration.

5. *How to plan and executes the changes from the current configuration to a newly produced configuration, with no need to write by hand low-level platform-specific reconfiguration scripts?*
   We rely on existing reflection mechanisms to improve them by providing a higher degree of validation and confidence during the dynamic adaptation process. The running system is abstracted as a model, which currently captures the structure of the system. If a model produced (by aspect weaving) by the adaptation chain is valid, the migration process is then fully automated. We compare the current reflection model to the new model to determine a sequence of atomic actions needed to actually reconfigure the system, thus preventing architects from writing low level scripts using the reflection API of the platform.

The approach presented in this thesis is effectively supported by tools developed in the context of the DiVA project. These tools are currently used by industrial partners to implement their case studies. Preliminary feedback from our industrial partners are promising and indicate that the approach and associated tools are actually usable in an industrial context.

Chapter 9

Perspectives

Contents

9.1 Dual-View Aspects to Support Behavioral Adaptation and Validation

The recent emergence of Aspect-Oriented Multi-View Modeling [85] would allows us to seamlessly extend our approach with more advanced validation and adaptation capabilities. The basic idea would be to refine the variants of the variability model using dual-view aspects. Each aspects would have:

- **A Structural view**, describing the components and bindings of the aspects, which is currently the view we have.

- **A Behavioral view**, describing the behavior of the aspects, probably using a state-based model.

This would allows us to build the whole structure and the whole behavior of each configuration in an incremental. In addition to the structural checking, we could perform behavioral validation: detection of live-locks and dead-locks, verification of temporal properties [166], etc. In addition, this would make it possible to dynamically modify the behavior of components [8], in a controlled way. However, we should probably use a dedicated aspect model weaver in order to fully consider the semantics of behavioral models [86, 161].

9.2   Bootstrapping the Adaptation Loop to Support Evolution

As we have seen in Chapter 6, the reference architecture supporting our different contributions is implemented as a component-based system. We actually use a causal link component to deploy our adaptation loop, which is responsible for dynamically adapting the business system. It would be technically straightforward to use a whole adaptation loop (and not just the causal link component) to deploy, and even dynamically adapt, another adaptation loop. This would lead to a 3-layered architecture, as illustrated in Figure 9.1:

- **Business Layer** (right): This layer describes the architecture of the business application. This layer is managed by the adaptation layer, which can dynamically reconfigure the business architecture (addition/removal of components/bindings) depending on the monitored context.

- **Adaptation Layer** (middle): This layer is responsible for managing the business layer. It basically consists of a MAPE loop we usually found in most adaptive systems [81, 82]. This MAPE loop is fed with knowledge (mainly variability and reasoning models) to determine when and how to adapt the business architecture. The novelty is that this loop is dynamically adaptive and managed by the evolution layer.

- **Evolution (Meta-Adaptation) Layer** (left): This layer is responsible for managing the adaptation layer *i.e.*, to reconfigure the MAPE loop that manages the business system. This layer is also a MAPE loop, but it does not evolve at runtime[1]. It monitors the requirements of the business architecture. More precisely, it observes the knowledge of the adaptation layer and can decide to dynamically reconfigure the adaptation layer to take full advantage of this new knowledge.

We envision that the ability to dynamically adapt an adaptation loop would ease the evolution of a DAS. For example, let us consider that the D-CRM system has been designed only with goal optimizations (because the adaptation is mainly driven by QoS properties). In this initial version, it does not make sense to use an ECA reasoner in the adaptation loop, since this component would be totally useless. We thus only deploy a goal- based reasoner in the adaptation loop.

After a few weeks, many users of the D-CRM complain because they cannot always access the resources of the server when they are using the external WiFi access points, the guest WiFi when they visit clients, or public WiFi access points. This is due to the use of a VPN over IPSec, whose packets are sometimes black-listed by firewalls.

The communication with the server is an important requirement of the D-CRM. The engineers developing and maintaining the D-CRM thus install a VPN over SSL (Secure

---

[1]Even if it could of course be possible to define a meta-meta-adaptation layer.
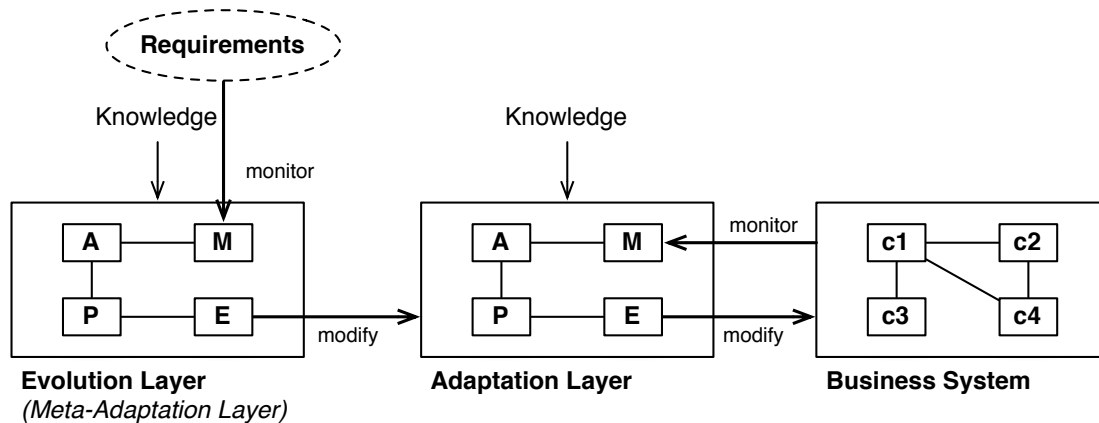
Figure 9.1: An Adaptation Loop managing Another Adaptation Loop, which itself manages a business system

Sockets Layer) server. The security feature related to the communication is now an alternative: IPSec or (exclusive) SSL. However, the choice of one or the other variant is not driven by QoS properties. By default, the engineers prefer the IPSec solution, since it is more customizable. This naturally leads to the following rules:

- if connection requested then use VPN over IPSec

- if VPN over IPSec fails then use VPN over SSL

The rules can be very easily expressed with the ECA paradigm. However, no ECA reasoner is available in the current adaptation rule. Since the adaptation layer is managed by the evolution layer, the introduction of ECA rules automatically triggers the introduction of an ECA reasoner in the adaptation layer. The additional rules (expressed in another paradigm) can thus be seamlessly integrated.

9.3 Advanced Model Composition to Domain-Driven Dynamic Adaptation

One of the main contributions of this thesis is to adopt a DSPL approach [69, 105, 104] and to refine and compose the dynamic features of the system using Aspect-Oriented Modeling techniques [88, 77]. However, aspect models are defined in terms of components and bindings, which is still a rather low level of abstraction.

We have investigated more advanced composition mechanisms to raise the level of abstraction in the context of dynamic adaptation [110].

For example Figure 9.2 shows how we can map access control concepts to architectural concepts. The composition of the access control concepts with the architectural concepts can be implemented in Kermeta.
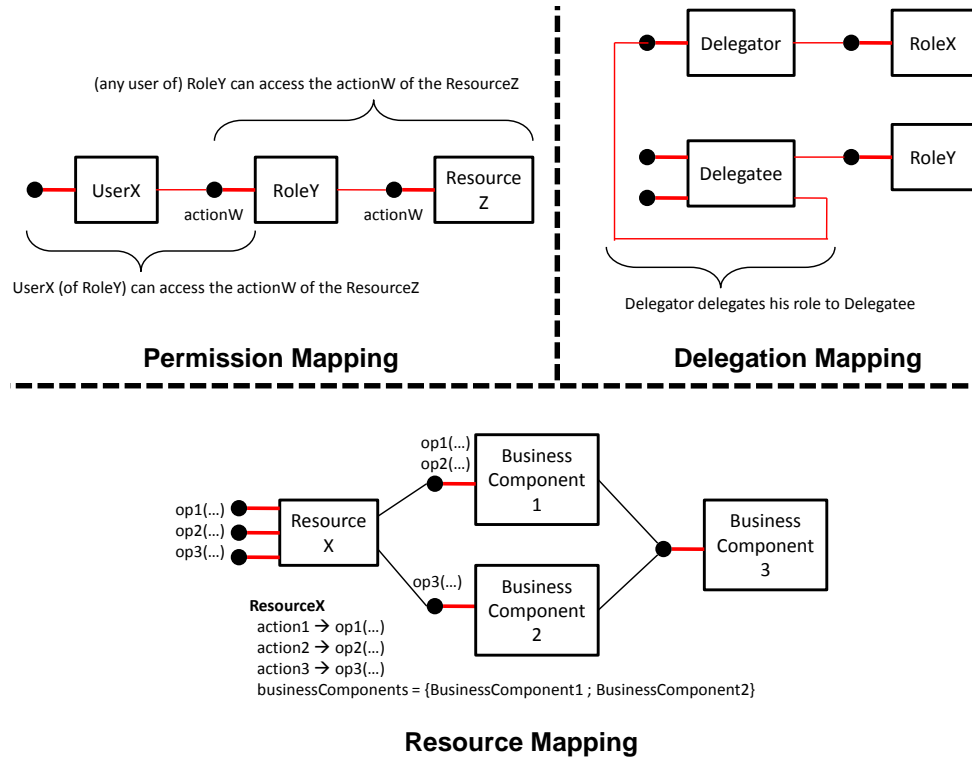

Figure 9.2: Mapping Permissions, Delegations and Resources (Access Control concepts) to Architectural Concepts

The basic idea is to generate several layers of components, whose implementation is straightforward and fully generated, between the users and the resources depending on an access control policy. These components and bindings makes is possible to administrate very finely and dynamically the access control policy.

At the end, we still rely on reconfiguration of components and bindings. However, the architecture and the reconfigurations are totally inferred from higher-level specification (access control policies). Other domains such as GUI or workflows could probably be mapped to architecture, making it possible to apply the same kind of approaches to develop more flexible GUIs and workflows.

9.4 From Requirements to Software and Hardware Dynamic Adaptation

In the context of the DiVA project, we are currently developing a tool supported methodology, which integrates the work carried out in this thesis, for engineering adaptive systems. This methodology starts from the requirements, expressed in natural language (plain English). Lancaster University has developed some tools (EA-Miner [137, 138] and ArborCraft [4, 117]) that process natural language to identify variability concerns and early aspects[2]. From this analysis, it is possible to populate initial design-models that should then be completed by designers. The use of other requirement engineering techniques, such as goal models [62, 163, 36] could probably allows populating initial design models more thoroughly by identifying adaptation rules and goals the system should optimize.

In addition to bridging the gap between requirement and design, we also want to bridge the gap between software adaptation and hardware adaptation. Indeed, modern hardware system also provide techniques for reconfiguration. For example, FPGA (Field-programmable gate array) [72][3] can be configured and reconfigured via an HDL (Hardware Description Language), at a very fine grain (*i.e.* wiring of AND/OR logic gates). Having a an approach that leverage both software and hardware reconfiguration could be really useful in the context of sensor networks *e.g.*, to predict flood [74, 75] or to prospect offshore resources, etc.

---

[2]See Early Aspects workshops: http://www.early-aspects.net/

[3]see also the FPGA conferences since 2001, and the websites of the two main manufacturers of FPGA: Xilinx http://www.xilinx.com/ and Altera http://www.altera.com/

List of Figures

List of Tables

Bibliography

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

[2] The Fractal Project. OW2 Consortium. Website: http://fractal.ow2.org/.

[3] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK programming language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. An Exploratory Study of Information Retrieval Techniques in Domain Analysis. In *SPLC'08: 12th Software Product Line Conference*, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[5] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR'04: 8th International Conference on Software Reuse: Methods, Techniques and Tools*, pages 141–156, Madrid, Spain, 2004.

[6] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM.

[7] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer, 2006.

[8] 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.

[9] E. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *ICSE'04: 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA'05: 2nd European Workshop on Software Architecture, Pisa, Italy*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[11] J. Bayer, S. Grard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J.-P. Thibault, and T. Widen. *Software Product Lines*, chapter Consolidated Product Line Variability Modeling, pages 195–242. Number ISBN: 978-3-540-33252-7. Springer Verlag, 2006.

[12] 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 at ECOOP'05, Glasgow, UK, July 15, 2005*, pages 73–82.

[13] N. Bencomo, G. Blair, C. Flores, and P. Sawyer. Reflective Component-based Technologies to Support Dynamic Variability. In *VaMoS'08: 2nd Int. Workshop on Variability Modeling of Software-intensive Systems*, Essen, Germany, January 2008.

[14] N. Bencomo, P. Grace, C. Flores, and D. Hughesand G. Blair. Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. In *ICSE 2008 - Formal Research Demonstrations Track*, 2008.

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

[16] Nelly Bencomo. On the Use of Software Models during Software Execution. In *MISE'09: Proceedings of the Workshop on Modeling in Software Engineering, at ICSE'09*, 2009.

[17] Nelly Bencomo and Gordon Blair. Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems. 5525:183–200, 2009.

[18] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[19] Gordon Blair, Geoff Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In Seitz J. Davies N.A.J., Raymond K., editor, *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 91–206, The Lake District, UK, 1998.

[20] Gordon Blair, Geoff Coulson, Jo Ueyama, Kevin Lee, and Ackbar Joolia. Opencom v2: A component model for building systems software. In *IASTED Software Engineering and Applications*, USA, 2004.

[21] Gordon S. Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based architecture: the fractal initiative. *Annales des Télécommunications*, 64(1-2):1–4, 2009.

[22] J. Boardman and B. Sauser. System of systems - the meaning of of. *System of Systems Engineering*, 0:6 pp.–, 2006.

[23] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. CLOS in context: the shape of the design space. pages 29–61, 1993.

[24] J.A. Bondy and U.S.R. Murty. *Graph theory with applications*. MacMillan London, 1976.

[25] Sara Bouchenak, Nol De Palma, Daniel Hagimont, and Christophe Taton. Autonomic Management of Clustered Applications. In *IEEE International Conference on Cluster Computing (Cluster 2006)*, Barcelona, Spain, September 2006.

[26] L. Bouzonnet, P.C. David, T. Ledoux, and N. Loriant. WildCAT A Generic Framework for Context-Aware Applications. http://wildcat.ow2.org/.

[27] Fabienne Boyer, Noel Palma, Olivier Gruber, Sylvain Sicard, and Jean-Bernard Stefani. A self-repair architecture for cluster systems. pages 124–147, 2009.

[28] 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.

[29] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The FRACTAL Component Model and its Support in Java. *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[30] 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.

[31] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *Computer*, 42:37–43, 2009.

[32] 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, Valencia, Spain, 2009. IEEE Computer Society.

[33] F. Chauvel, O. Barais, I. Borne, and J-M. Jézéquel. Composition of Qualitative Adaptation Policies. In *ASE'08: 23rd IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, Italy, sep 2008.

[34] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty . In *MODELS'09: ACM/IEEE 12th International Conference on Model-Driven Engineering Languages and Systems*, Denver, Colorado, USA, oct 2009.

[35] 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. 5525:1–26, 2009.

[36] Betty H. C. Cheng, Peter Sawyer, Nelly Bencomo, and Jon Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In Andy Schrr and Bran Selic, editors, *MoDELS'09: ACM/IEEE 12th International Conference of Model-Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 468–483, Denver, Colorado, USA, 2009. Springer.

[37] 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*, pages 2–8, New York, NY, USA, 2006. ACM.

[38] P. Clements and L. Northrop. *Software product lines: practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[39] P.C. David and T. 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.

[40] P.C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *SC'06: 5th Int. Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97, Vienna, Austria, 2006.

[41] P.C. David and T. Ledoux. Safe Dynamic Reconfigurations of Fractal Architectures with FScript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006.

[42] 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, Oslo, Norway, 2008.

[43] Vegard Delhen and Franck Fleurey, editors. *DiVA Deliverable D2.1: Transformation Framework*. 2009.

[44] E.A. Emerson. Temporal and modal logic. *Handbook of theoretical computer science*, 8:995–1072, 1990.

[45] Betty Cheng *et al.* Proceedings of the international workshops on software engineering for adaptive and self-managing systems(2005-2009) at icse, 2005-2009.

[46] N. Ferry, S. Lavirotte, J-Y. Tigli, and G. Rey et M. Riveill. Context Adaptative Systems based on Horizontal Architecture for Ubiquitous Computing . *Mobility'09: 6th International Conference on Mobile Technology, Applications and Systems, Nice, France*, September 2009.

[47] Nicolas Ferry, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Jean-Yves Tigli, and Michel Riveill. Models at Runtime: Service for Device Composition and Adaptation. In *4th International Workshop Models@run.time at Models 2009(MRT'09) AR=31%*, October 2009.

[48] Eduardo Figueiredo, Nélio Cacho, Claudio SantAnna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE'08: 30th International Conference on Software Engineering*, pages 261–270, Leipzig, Germany, may 2008. ACM.

[49] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach For Automatic Model Composition. In *AOM@MoDELS'07: 11th Int. Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.

[50] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J-M. Jézéquel. Modeling and Validating Dynamic Adaptation. In *3rd International Workshop on Models@Runtime, at MODELS'08*, Toulouse, France, oct 2008.

[51] F. Fleurey and A. Solberg. A Domain Specific Modeling Language supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *MODELS'09:*

ACM/IEEE 12th International Conference on Model-Driven Engineering Languages and Systems, Denver, Colorado, USA, oct 2009.

[52] J. Floch. Theory of Adaptation. *Deliverable D2.2, MADAM project*, 2006.

[53] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.

[54] Charles Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[55] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing Support for Model Composition in Metamodels. In *EDOC'07: 11th Int. Enterprise Computing Conf.*, 2007.

[56] Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, Annapolis, MD, USA, 2007.

[57] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[58] Erann Gat. Three-layer architectures. pages 195–210, 1998.

[59] Nikolaos Georgantas, Sonia Ben Mokhtar, Yerom-David Bromberg, Valerie Issarny, Jarmo Kalaoja, Julia Kantarovitch, Anne Gerodolle, and Ron Mevissen. The Amigo Service Architecture for the Open Networked Home Environment. In *WICSA'05: 5th Working IEEE/IFIP Conference on Software Architecture*, pages 295–296, Washington, DC, USA, 2005. IEEE Computer Society.

[60] J. Georgas, A. van der Hoek, and R. Taylor. Using Architectural Models to Manage and Visualize Runtime Adaptation. *Computer*, 42(9):52–60, 2009.

[61] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization . *Software and Systems Modeling (SoSyM)*, 8(1), 3 2009.

[62] H. Goldsby, P. Sawyer, N. Bencomo, B.H.C. Cheng, and D. Hughes. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *ECBS'08: 15th IEEE International Conference on the Engineering of Computer Based Systems*, pages 36–45, Belfast, Northern Ireland, 2008. IEEE Computer Society.

[63] Greg Goth. Ultralarge systems: Redefining software engineering? *IEEE Software*, 25:91–94, 2008.

[64] I. Groher and M. Voelter. Using Aspects to Model Product Line Variability. In *EA@SPLC'08: 13th International Workshop on Early Aspects at SPLC*, Limerick, Ireland, 2008.

[65] Roy Grønmo, Stein Krogdahl, and Birger Møller-Pedersen. A Collection Operator for Graph Transformation. In *ICMT'09: 2nd International Conference on Theory and Practice of Model Transformations*, pages 67–82, Berlin, Heidelberg, 2009. Springer-Verlag.

[66] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[67] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[68] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Proceedings of the international workshops on dynamic software product lines (2007-2009) at splc, 2007-2009.

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

[70] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *SPLC'06: 10th Int. Software Product Line Conf.*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[71] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A Case Study in Goal-Driven Architectural Adaptation. 5525:109–127, 2009.

[72] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. Dynamic hardware plugins in an fpga with partial run-time reconfiguration. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 343–348, New York, NY, USA, 2002. ACM.

[73] Vincent Hourdin, Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. SLCA, composite services for ubiquitous computing. In *Mobility '08: 5th International Conference on Mobile Technology, Applications, and Systems*, pages 1–8, Yilan, Taiwan, 2008. ACM.

[74] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. An experiment with reflective middleware to support grid-based flood-monitoring. *Concurr. Comput. : Pract. Exper.*, 20(11):1303–1316, 2008.

[75] 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 *4th International Workshop on Mobile Distributed Computing (MDC'06)*, Niagara Falls, USA, 2006.

[76] Valérie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Françoise Sailhan, Rafik Chibout, Nicole Levy, and Angel Talamona. Developing Ambient Intelligence Systems: A Solution based on Web Services. *Automated Software Engineering*, 12(1):101–137, 2005.

[77] P.K. Jayaraman, J. Whittle, A.M. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.

[78] P.K. Jayaraman, J. Whittle, A.M. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.

[79] F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2006. Springer Verlag.

[80] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *SPLC '07: 11th International Software Product Line Conference*, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[83] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. In *ECOOP'01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[84] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[85] Jörg Kienzle, Wisam Al Abed, and Klein Jacques. Aspect-Oriented Multi-View Modeling. In *AOSD '09: 8th ACM international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2009. ACM.

[86] J. Klein, L. Hélouet, and J-M. Jézéquel. Semantic-based weaving of scenarios. In *AOSD'06: 5th International Conference on Aspect-OrientedSoftware Development*, Bonn, Germany, 2006. ACM.

[87] George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, May 1995.

[88] Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, October 2007.

[89] Ph. Lahire and L. Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal Of Object Technology (JOT)*, 5(1):117–138, 2006.

[90] M. Leclercq, A. Erdem Ozcan, V. Quéma, and J.B. Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *ICSE'07: 29th Int. Conf. on SoftwareEngineering*, pages 209–219, Washington, DC, USA, 2007.

[91] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *ARM'07: 6th international workshop on Adaptive and reflective middleware*, pages 1–6, Newport Beach, CA, 2007. ACM.

[92] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable Dynamic Reconfigurations in a Reflective Component Model. In *CBSE'10: 13th International Symposium on Component Based Software Engineering*, Prague, Czech Republic, 23-25 June 2010, 2010.

[93] 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.

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

[95] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[96] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Using resemblance to support component reuse and evolution. In *SAVCBS'06: Conference on Specification and verification of component-based systems*, pages 49–56, New York, NY, USA, 2006. ACM.

[97] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT'04/FSE-12: 12th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 127–136, Newport Beach, CA, USA, 2004. ACM.

[98] Sonia Ben Mokhtar. *Semantic Middleware for Service-Oriented Pervasive Computing*. PhD thesis, Universit Pierre et Marie Curie - Paris 6, 2007.

[99] Sonia Ben Mokhtar, Nikolaos Georgantas, and Valérie Issarny. COCOA: COnversation-based service COmposition in pervAsive computing environments with QoS support. *Journal of Systems and Software*, 80(12):1941–1955, 2007.

[100] B. Morin, O. Barais, and J. M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *VaMoS'08: 2nd Int. Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, January 2008.

[101] B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd Int. ECOOP'07 Workshop on Models and Aspects, Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.

[102] B. Morin, J.Klein, O. Barais, and J. M. Jézéquel. A Generic Weaver for Supporting Product Lines. In *EA@ICSE'08: Int. Workshop on Early Aspects*, Leipzig, Germany, May 2008.

[103] Brice Morin, Olivier Barais, and Jean-Marc Jézéquel. K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In *Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08*, Toulouse, France, oct 2008.

[104] Brice Morin, Olivier Barais, Jean-Marc Jzquel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[105] Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.

[106] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jzquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 08)*, Toulouse, France, October 2008.

[107] Brice Morin, Franck Fleurey, Grgory Nain, Olivier Barais, and Jean-Marc Jzquel. Aspect-Oriented Modeling to Support Dynamic Adaptation (Forum Demo). In

AOSD'10: 9th International Conference on Aspect-Oriented Software Development, St Malo, France, Mar 2010.

[108] Brice Morin, Jacques Klein, Jorg Kienzle, and Jean-Marc Jzquel. Flexible model element introduction policies for aspect-oriented modeling. In *Proceedings of ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*, Oslo, Norway, October 2010.

[109] Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel, Olivier Barais, and Jean-Marc Jzquel. Unifying Runtime Adaptation and Design Evolution. In *IEEE 9th International Conference on Computer and Information Technology (CIT'09)*, Xiamen, China, Oct 2009.

[110] Brice Morin, Tejeddine Mouelhi, Franck Fleurey, Yves Le Traon, Olivier Barais, and Jean-Marc Jézéquel. Security-Driven Model-Based Dynamic Adaptation. In *25nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, Antwerp, Belgium, September 2010.

[111] Brice Morin, Grgory Nain, Olivier Barais, and Jean-Marc Jzquel. Leveraging Models From Design-time to Runtime. A Live Demo. In *4th International Workshop on Models@Run.Time (at MODELS'09)*, Denver, Colorado, USA, Oct 2009.

[112] P.A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.

[113] Freddy Munoz and Benoit Baudry. Artificial Table Testing Dynamically Adaptive Systems. Research report inria-00365874, INRIA Bretagne Atlantique, http://hal.inria.fr/inria-00365874/en/, 2009.

[114] R. France N. Bencomo, G. Blair. Proceedings of the international workshops on models@run.time (2006-2009) at models, 2006-2009.

[115] G. Nain, F. Fouquet, B. Morin, O. Barais, and J-M. Jézéquel. Integrating IoT and IoS with a Component-Based approach. In *SEAA 2010: 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, Lille, France, 2010.

[116] Grgory Nain, Erwan Daubert, Olivier Barais, and Jean-Marc Jézéquel. Using MDE to Build a Schizofrenic Middleware for Home/Building Automation. In *In ServiceWave'08: Networked European Software & Services Initiative (NESSI) Conference*, Madrid, Spain, December 2008.

[117] Joost Noppen, Pim van den Broek, Nathan Weston, and Awais Rashid. Modelling Imperfect Product Line Requirements with Fuzzy Feature Diagrams. In *Va-*

MoS'09: 3rd International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, pages 93–102, 2009.

[118] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[119] D. Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[120] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling Automation and Flexibility in Product Derivation. In *SPLC'08: 12th International Software Product Line Conference*, pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society.

[121] Nicolas Pessemier, Olivier Barais, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A Three Level Framework for Adapting Component Based Architectures. In *WCAT'05@ECOOP: 2nd Workshop on Coordination and Adaptation Techniques for Software Entities*, jul 2005.

[122] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A Component-Based and Aspect-Oriented Model for Software Evolution. In *IJCAT'07: International Journal of Computer Applications in Technology, Special Issue on Concern-Oriented Software Evolution*, volume 4089 of *Lecture Notes in Computer Science*, page 259273, Vienna, Austria, mar 2006. Springer-Verlag.

[123] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A Model for Developing Component-based and Aspect-oriented Systems. In *SC'06: 5th International Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, page 259273, Vienna, Austria, mar 2006. Springer-Verlag.

[124] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A Safe Aspect-Oriented Programming Support for Component-Oriented Programming. In *WCOP'06@ECOOP: 11th International Workshop on Component-Oriented Programming*, volume 200611 of *Technical Report*, Nantes, France, jul 2006. Karlsruhe University.

[125] Nicolas Pessemier, Lionel Seinturier, and Laurence Duchien. Components ADL and AOP: Towards a Common Approach. In *RAM-SE'04@ECOOP: Workshop on Reflection, AOP and Meta-Data for Software Evolution*, Oslo, Norway, jun 2004.

[126] Sriplakich Prawee, Guillaume Waignier, and Anne-Françoise Le Meur. Enabling Dynamic Co-evolution of Models and Runtime Applications. In *32nd Annual IEEE International COMPSAC '08*, pages 1116 – 1121, Turku, Finlande, 2008.

[127] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of ECOOP'97*. Springer-LNCS, 1997.

[128] R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model Snippets. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, page 15, Nashville USA, October 2007.

[129] Awais Rashid and Ana Moreira. Domain Models Are NOT Aspect Free. In *MoDELS'06: 9th International Conference on Model Driven Engineering Languages and Systems*, pages 155–169, Genova, Italy, 2006.

[130] Y. R. 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. In Awais Rashid and Mehmet Aksit, editors, *Transaction on Aspect-Oriented Software Development*, volume vol 3880 of *Lecture Notes in Computer Science*, pages 75–105. Springer, 2006.

[131] Jeff Rothenberg, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. The Nature of Modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.

[132] 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. 5525:164–182, 2009.

[133] 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.

[134] 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, 2008. Springer.

[135] S.J. Russell, P. Norvig, J.F. Canny, J. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, NJ, 1995.

[136] S. Masoud Sadjadi, Philip K. McKinley, and Betty H. C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM.

[137] Américo Sampaio, Ruzanna Chitchyan, Awais Rashid, and Paul Rayson. EA-Miner: a tool for automating aspect-oriented requirements identification. In *ASE'05: 20th IEEE/ACM international Conference on Automated software engineering*, pages 352–355, New York, NY, USA, 2005. ACM.

[138] Américo Sampaio, Awais Rashid, Ruzanna Chitchyan, and Paul Rayson. EA-Miner: Towards Automation in Aspect-Oriented Requirements Engineering. *T. Aspect-Oriented Software Development*, 3:4–39, 2007.

[139] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), February 2006.

[140] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. In *SCC'09: IEEE International Conference on Services Computing*, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society.

[141] Sagar Sen, Benoit Baudry, and Doina Precup. Partial Model Completion in Model-Driven Engineering using Constraint Logic Programming. In *INAP'07: 16th International Conference on Applications of Declarative Programming and Knowledge Management*, 2007.

[142] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag.

[143] 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, *4th International Workshop on Models@run.time at MODELS 2009, Denver, Colorado, USA*, volume 509 of *CEUR Workshop Proceedings*, pages 1–10, 10 2009.

[144] Jim Steel. *Typage de modles*. PhD thesis, Universit de Rennes 1, April 2007. In English.

[145] Jim Steel and Jean-Marc Jzquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, December 2007.

[146] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge (MA), 1986.

[147] DiVA Consortium (FP7 STREP). Diva: Dynamic variability in complex adaptive systems, 2008-2011.

[148] Bholanathsingh 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*, pages 1–6, New York, NY, USA, 2007. ACM.

[149] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From Goals to Components: A Combined Approach to Self-Management. In *SEAMS'08: International workshop on Software engineering for adaptive and self-managing systems at ICSE'08*, Leipzig, Germany, 2008.

[150] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance*, pages 446–453, 2004.

[151] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE'99: 21st international conference on Software engineering*, pages 107–119, New York, NY, USA, 1999. ACM.

[152] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari, and Michel Riveill. WComp Middleware for Ubiquitous Computing: Aspects and Composite Event-based Web Services. *Annals of Telecommunications (AoT)*, 64(3-4):197–214, April 2009.

[153] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, and Michel Riveill. Context-aware Authorisation in Highly Dynamic Environments. *International Journal of Computer Science Issues (IJCSI)*, 4, September 2009.

[154] Sun Tzu (translated by Samuel B. Griffith). *The Art of War*. Oxford University, 1963.

[155] Augusto Visintin. *Differential models of hysteresis*. Springer, 2004.

[156] 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, *4th International Workshop on Models@run.time at MODELS 2009, Denver, Colorado, USA*, volume 509 of *CEUR Workshop Proceedings*, pages 1–10, 10 2009.

[157] 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.

[158] Guillaume Waignier, Sriplakich Prawee, Anne-Françoise Le Meur, and Laurence Duchien. A Framework for Bridging the Gap Between Design and Runtime Debugging of Component-Based Applications. In *3rd International Workshop on Models@runtime*, Toulouse France, 2008.

[159] Guillaume Waignier, Sriplakich Prawee, Anne-Françoise Le Meur, and Laurence Duchien. A Model-Based Framework for Statically and Dynamically Checking Component Interactions. In *ACM/IEEE 11th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2008) Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 371–385, Toulouse France, 2008.

[160] J. Whittle and P. Jayaraman. MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. In *AOM@MoDELS'07: 11th Int. Workshop on Aspect-Oriented Modeling*, Nashville USA, Oct 2007.

[161] J. Whittle, A. Moreira, J. Araujo, P. Jayaraman, A. Elkhodary, and R. Rabbi. An Expressive Aspect Composition Language for UML State Diagrams. In *MoDELS'07, ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, LNCS, Nashville, USA, 2007. Springer.

[162] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. Mata: A unified approach for composing uml aspect models based on graph transformation. *T. Aspect-Oriented Software Development VI*, 6:191–237, 2009.

[163] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *RE'09: 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*, pages 79–88, 2009.

[164] Apostolos Zarras, Manel Fredj, Nikolaos Georgantas, and Valérie Issarny. Engineering Reconfigurable Distributed Software Systems: Issues Arising for Pervasive Computing. In Michael J. Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, volume 4157 of *Lecture Notes in Computer Science*, pages 364–386. Springer, 2006.

[165] 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, 2006. ACM Press.

[166] Ji Zhang and Betty H. C. Cheng. Using temporal logic to specify adaptive program semantics. volume 79, pages 1361–1369, 2006.

[167] Ji Zhang, Betty H. C. Cheng, Zhenxiao Yang, and Philip K. McKinley. Enabling safe dynamic component-based software adaptation. In Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky, editors, *WADS*, volume 3549 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2004.

[168] 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, 2009. ACM.

[169] Tewfik Ziadi and Jean-Marc Jézéquel. *Software Product Lines*, chapter Product Line Engineering with the UML: Deriving Products, pages 557–586. Number ISBN: 978-3-540-33252-7. Springer Verlag, 2006.

Appendix A

Implementation Details

Contents

A.1   Mapping the SCA metamodel to the ART metamodel

The following script written in Kermeta [112] shows how we map the SCA metamodel to the ART core architecture metamodel. We first require both metamodels. Note that we are working in the package sca. Kermeta offers a seamless weaving engine that allows designers to extend a metamodel with additional elements: attributes, references, contracts (invariants and pre/post conditions), operations (or implement an already existing abstract operation). Here, we re-open the Wire meta-class (from the SCA metamodel) in order to:

- Add a new reference to a transmission binding (from the ART core architectural metamodel)

- Add a visit operation to instantiate this newly introduced reference and set its properties, using the woven references of already visited elements.

```
1  package sca;
2
3  require kermeta
```

```
4   require "http://www.osoa.org/xmlns/sca/1.0" //SCA metamodel
5   require "http://art" //ART metamodel
6   /**
7    * Opens the SCA::Wire metaclass to map it
8    * to  art :: TransmissionBinding
9    */
10  aspect class Wire {
11
12    reference artBinding : TransmissionBinding [1..1]
13
14    operation visit() is
15    do
16      artBinding := TransmissionBinding.new
17      artBinding.server := self.target2.artPort
18      artBinding.client := self.source2.artPort
19      artBinding.serverInstance := self.target2.container
20          .asType(Component).artComponent
21      self.source2.container.asType(Component).artComponent
22          .binding.add(artBinding)
23    end
24  }
```

The part of the SCA metamodel relevant to ART is mapped to the ART (architecture) metamodel in the same way. Then, it is possible to transform an architectural model conforming to the SCA metamodel to an architectural metamodel conforming to the ART Core metamodel.

A.2   Compilation of Aspect Models

This Appendix gives some implementation details related to the compilation of aspect models in Java + Drools. As explained in Section 4.6.2, the compilation process is performed in two steps:

1. A meta-code generator takes a domain metamodel as input in order to generate a domain-specific compiler (Java + Drools code generator) for SmartAdapters. This meta-code generator mainly consists of a code template written in KET (Kermeta Emitter Template). For each meta-class of the domain metamodel, the KET template generates a Kermeta aspect that re-opens this meta-class in order to weave two methods.

2. A domain-specific compiler takes an aspect model as input an generates the Java + Drools code corresponding to this aspect. This domain-specific code generator is fully generated and corresponds to all the Kermeta aspects.

For example, the following (generated) script re-opens the `PrimitiveInstance` meta-class defined in our architectural metamodel in order to weave to methods, responsible

for generating the Drools and Java code associated to an aspect model. Basically, these two methods define a visitor in 2 pass: the first one creates all the elements of the advice model and the second one sets all the references among these elements. The scope of the advice model elements is handled in the first pass. All the generated Kermeta aspects (M2 level) make it possible to compile SmartAdapters aspects (M1 level), such as the one described in Appendix A.3 and Appendix A.4. These two Appendices further discuss the way we handle the notion of uniqueness and scoped uniqueness.

Note that the KET template used to generate this script is almost similar to this script. This KET template simply takes an EClass as input (*i.e.,* a meta-class). Indeed, the only information we need in the template are the name of the meta-class (*e.g.,* `Primitive-Instance`), its qualified name (*e.g.,* `art.instance.PrimitiveInstance`) and other strings that can be inferred (*e.g.,* `art.instance.InstanceFactory.eINSTANCE.create-PrimitiveInstance()`) from these pieces of information, as well as the references of the meta-class (to visit all the contained elements in pass1 and pass2, and to set these references in pass2).

```
1   /**
2    * The PrimitiveInstance meta−class is re−opened to weave
3    * the methods needed to generate the Drools+Java code ie,
4    * needed to compile aspect models.
5    */
6   aspect class PrimitiveInstance {
7     /**
8      * pass1create is responsible for creating the elements of the advice model.
9      * In particular, it manages the (scoped) uniqueness of these elements
10      */
11    method pass1create(ctx:Context):Void is do
12      //Associates a unique string to this (self) element
13      var na : String init ctx.getGenerateName
14      ctx.cache.put(self,na)
15
16      //if this element is unique
17      if self.ownedTags.exists{tag | "unique".equals(tag.name)} then
18        ctx.res.append("art.instance.PrimitiveInstance" +" "+ na + " = (art.
              instance.PrimitiveInstance) uniqueobjects.get(\""+na+"\");")
19        ctx.res.append("if ("+" "+ na + " " + "== null){")
20        ctx.res.append(" "+na + " = art.instance.InstanceFactory.eINSTANCE.
              createPrimitiveInstance();")
21        ctx.res.append(" uniqueobjects.put(\""+na+"\","+ na+");")
22        ctx.res.append("}")
23      else
24        //else if this element is unique with scope
25        if self.ownedTags.exists{tag | "uniqueWithScope".equals(tag.name)} then
26          var scopeName : String init "scope_"+ctx.getGenerateName
27          ctx.res.append("Set<EObject> "+scopeName+" = new HashSet<EObject>();")
28          self.ownedTags.detect{tag | "uniqueWithScope".equals(tag.name)}
29            .~value.split(" ").each{elt |
```

```
30              ctx.res.append(scopeName+".add("+elt+");")
31            }
32
33        ctx.res.append("if (uniqueObjectsWithScope.get("+scopeName+") == null){")
34        ctx.res.append(" uniqueObjectsWithScope.put("+scopeName+", new Hashtable<
               String, EObject>());")
35        ctx.res.append("}")
36
37        ctx.res.append("art.instance.PrimitiveInstance" +" "+ na + " = (art.
               instance.PrimitiveInstance) uniqueObjectsWithScope.get("+scopeName+"
               ).get(\""+na+"\");")
38        ctx.res.append("if ("+" "+ na + " " + "== null){")
39        ctx.res.append(" "+na + " = art.instance.InstanceFactory.eINSTANCE.
               createPrimitiveInstance();")
40        ctx.res.append(" uniqueobjectsWithScope.get("+scopeName+").put(\""+na+"
               \","+ na+");")
41        ctx.res.append("}")
42      else  //if this element is per join point
43        ctx.res.append("art.instance.PrimitiveInstance" +" "+ na + " = art.
               instance.InstanceFactory.eINSTANCE.createPrimitiveInstance();")
44      end
45    end
46
47    //visiting all the contained elements
48    if self.~attribute.size >0 then
49      self.~attribute.each{c|c.pass1create(ctx)}
50    end
51    //...
52  end
53
54  /**
55   * pass2set is responsible for linking together the (already created) elements of the advice model
56   */
57  method pass2set(ctx:Context):Void is do
58    if (not self.name.isVoid()) then
59      ctx.res.append(ctx.cache.getValue(self) + ".set"+"Name"+"(\""+self.name.
               toString+"\");")
60    end
61    //...
62
63    //visiting all the contained elements
64    if self.~attribute.size >0 then
65      self.~attribute.each{c|c.pass2set(ctx)}
66    end
67    //...
68  end
69 }
```

A.3 Log Aspect compiled into Drools code when logger is unique

This script presents the log aspect compiled into Java/Drools code, in the case where the logger component is unique. The only modifications made to the raw generated script to improve readability are: the renaming of some variables, the introduction of comments, the removal of casts and the removal of the lines not relevant for the discussion.

Each application of the advice (then clause, Line 14-44) is independent from the previous ones. For example, if the pointcut (when clause, Line 6-12) matches several times in a base model, it is not possible to directly know if an element from the advice has already been created in a previous application of the aspect. This is an issue for the implementation of the notion of uniqueness.

However, it is possible to declare global variables (initialized in the Java code calling the Drools engine with the script), which allows to share data between different application of the advice. We thus use the `uniqueObjects` map as the structure to manage the uniqueness of advice elements. Basically, each advice element is identified by a unique string. Before creating a unique element, we check if this element already exists in the `uniqueObjects` global map. It this element already exists, it is reused. Otherwise, it is created and stored in the global map.

```
 1   global Map<String, EObject> uniqueObjects;
 2   global Map<Set<EObject>,Map<String, EObject>> uniqueObjectsWithScope;
 3
 4   rule "LoggingAspect"
 5
 6   when //Pointcut
 7     $logService: art.type.Service(name == "org.slf4j.Logger")
 8     $requiredLogPort: art.type.Port(role == "client",service == $logService)
 9     $anyType: art.type.PrimitiveType(port contains $requiredLogPort)
10     $anyComponent: art.instance.PrimitiveInstance(type == $anyType)
11     $anyComposite: art.instance.CompositeInstance(subComponent contains
             $anyComponent)
12     $anySystem: art.System(root == $anyComposite)
13
14   then
15     /*
16      * Creation of Advice model elements
17      */
18     //DefaultLogger component (unique)
19     art.instance.PrimitiveInstance logComponent = uniqueObjects.get("logComponent
             ");
20     if (logComponent == null){
21       logComponent = InstanceFactory.eINSTANCE.createPrimitiveInstance();
22       uniqueObjects.put("logComponent",logComponent);
23     }
24     //Binding to the DefaultLogger component (per join point)
```

```
25   art.instance.TransmissionBinding binding = InstanceFactory.eINSTANCE.
          createTransmissionBinding();
26   ...
27
28   /*
29    * Setting the references of Advice model elements
30    */
31   logComponent.setName("DefaultLogger");
32   logComponent.setType(s7);
33   logComponent.setImplem(s3);
34   ...
35
36   /*
37    * Actual Weaving
38    */
39   $anySystem.getServices().add(uniqueObjects.get("s6"));
40   $anySystem.getTypes().add(uniqueObjects.get("s7"));
41   $anyComposite.getSubComponent().add(uniqueObjects.get("logComponent"));
42   binding.setClient($requiredLogPort);
43   $anyComponent.getBinding().add(binding);
44   end
```

A.4   Log Aspect compiled into Drools code when logger is unique with scope

This script presents the log aspect compiled into Java/Drools code, in the case where the logger component is unique within the scope of a composite component. The only modifications made to the raw generated script to improve readability are: the renaming of some variables, the introduction of comments, the removal of casts and the removal of the lines not relevant for the discussion.

In order to deal with the notion of scoped uniqueness, we also use a global map: uniqueObjectsWithScope. The key of this map is a set of EObject (model elements) from the base model. The basic idea is to instantiate some advice elements only once for a given zone of the base model. In Java, the hash code of a set is computed as the sum of all the hash code of the elements contained in this set. It thus can be possible that two different sets (containing different elements) have the same hash code. While this is not recommended (for performance issues), a Map can handle this case and retrieve the right values even if two (different) keys have the same hash code. Indeed, the containsKey(Object key) method "returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))". Finally two sets are equals if "the two sets have the same size, and every member of the specified set is contained in this set (or equivalently, every member of this set is contained in the specified set)." This is exactly what we need to implement the scoped uniqueness.

```
1  global Map<String, EObject> uniqueObjects;
2  global Map<Set<EObject>,Map<String, EObject>> uniqueObjectsWithScope;
3
4  rule "LoggingAspect"
5
6  when //Pointcut
7    //Same as previous
8
9  then
10   /*
11    * Init of the structure managing the scopes
12    */
13   Set<EObject> compositeScope = new HashSet<EObject>();
14   compositeScope.add($anyComposite);
15   if (uniqueObjectsWithScope.get(compositeScope) == null){
16     uniqueObjectsWithScope.put(compositeScope, new Hashtable<String, EObject>())
           ;
17   }
18
19   /*
20    * Creation of Advice model elements
21    */
22   //DefaultLogger component (unique with scope)
23   art.instance.PrimitiveInstance logComponent = uniqueObjectsWithScope.get(
           compositeScope).get("logComponent");
24   if (logComponent == null){
25     logComponent = InstanceFactory.eINSTANCE.createPrimitiveInstance();
26     uniqueObjectsWithScope.get(compositeScope).put("logComponent",logComponent);
27   }
28   ...
29
30   /*
31    * Actual Weaving
32    */
33   $anyComposite.getSubComponent().add(uniqueObjectsWithScope.get(compositeScope
           ).get("logComponent"));
34    ...
35  end
```

A.5   Code Template to Generate Context Simulator

```
1  <%@ket
2  package="diva::monitoring"
3  require="platform:/lookup/diva.model/model/DiVA.ecore"
4  using="diva"
5  isAspectClass="false"
6  class="InteractiveContextSimulatorGenerator"
7  ismethod="false"operation="generate"
```

```
8    parameters="vm:VariabilityModel"
9    %>
10   package diva.runtime.monitoring.stub;
11
12   import java.awt.Component;
13   import ...
14
15   import javax.swing.JCheckBox;
16   import ...
17
18   import diva_context.Context;
19   import diva_context.ContextElement;
20   import diva_context.Diva_contextFactory;
21
22   /**
23    * This class allows simulating the context via a simple GUI.
24    * @author bmorin (Brice.Morin@inria.fr)
25    * This file was generated by KET: Kermeta Emitter Template in the
26    * context of the DiVA project
27    * see www.kermeta.org and www.ict−diva.eu
28    */
29   public class InteractiveContextSimulator implements ActionListener,
          ItemListener{
30
31    private JFrame frame;
32    private Context context;
33
34    public void initContext(){
35      context = Diva_contextFactory.eINSTANCE.createContext();
36    }
37
38    public void init(){
39      frame = new JFrame("Interactive Context Simulator");
40      frame.setLayout(new GridBagLayout());
41
42      GridBagConstraints c = new GridBagConstraints();
43      c.gridwidth = 1;
44      <%var y : Integer init 0%>
45      <%vm.context.each{cv | %>
46        <%if cv.isKindOf(EnumVariable) then%>
47      <%var cvEnum : EnumVariable
48      cvEnum ?= cv%>
49
50      /*
51       * Enumeration variable <%=cvEnum.name%>
52       */
53      JLabel label<%=y.toString%> = new JLabel();
54      label<%=y.toString%>.setText("<%=cvEnum.name%>");
55      c.gridx = 0;
56      c.gridy = <%=y.toString%>;
```

```
57      c.anchor = GridBagConstraints.WEST;
58      frame.add(label<%=y.toString%>, c);
59
60      JComboBox combo<%=y.toString%> = new JComboBox();
61      combo<%=y.toString%>.setName("<%=cvEnum.name%>");
62      <%cvEnum.literal.each{l | %>
63      combo<%=y.toString%>.addItem("<%=l.name%>");
64      <%}%>
65      c.gridx = 1;
66      c.gridy = <%=y.toString%>;
67      c.anchor = GridBagConstraints.EAST;
68      frame.add(combo<%=y.toString%>, c);
69      combo<%=y.toString%>.addActionListener(this);
70      combo<%=y.toString%>.addItemListener(this);
71        <%else%>
72      <%var cvBool : BooleanVariable
73      cvBool ?= cv%>
74
75      /*
76       * Boolean variable <%=cvEnum.name%>
77       */
78      JLabel label<%=y.toString%> = new JLabel();
79      label<%=y.toString%>.setText("<%=cvBool.name%>");
80      c.gridx = 0;
81      c.gridy = <%=y.toString%>;
82      c.anchor = GridBagConstraints.WEST;
83      frame.add(label<%=y.toString%>,c);
84
85      JCheckBox box<%=y.toString%> = new JCheckBox();
86      box<%=y.toString%>.setName("<%=cvBool.name%>");
87      c.gridx = 1;
88      c.gridy = <%=y.toString%>;
89      c.anchor = GridBagConstraints.EAST;
90      frame.add(box<%=y.toString%>,c);
91      box<%=y.toString%>.addActionListener(this);
92        <%end%>
93      <%y := y+1%>
94      <%}%>
95    }
96
97    /**
98     * Updates the boolean variables
99     */
100   public void actionPerformed(ActionEvent e) {
101     <%vm.context.select{cv | cv.isKindOf(BooleanVariable)}.each{cv | %>
102     if("<%=cv.name%>".equals(((Component)e.getSource()).getName())){
103       getContextElement("<%=cv.name%>").setCurrentValue("true");
104     }
105     <%}%>
106   }
```

```
107
108    /**
109     * Updates the enumeration variables
110     */
111    public void itemStateChanged(ItemEvent e) {
112      if(e.getStateChange() == ItemEvent.SELECTED){
113        <%vm.context.select{cv | cv.isKindOf(EnumVariable)}.each{cv | %>
114        <% var cvEnum : EnumVariable
115        cvEnum ?= cv%>
116        <%cvEnum.literal.each{l | %>
117        if("<%=l.name%>".equals(((String)e.getItem()))){
118          getContextElement("<%=cv.name%>").setCurrentValue("<%=l.name%>");
119        }
120        <%}%>
121        <%}%>
122      }
123    }
124
125    /**
126     *
127     * @param name
128     * @return gets the  context element whose name is name
129     * if  this  element does not exist,  it  is  created and
130     * added in the context  model.
131     */
132    private ContextElement getContextElement(String name) {
133      ContextElement ctxElt = null;
134
135      for(ContextElement ce : context.getElement()){
136        if(name.equals(ce.getName())){
137          ctxElt = ce;
138          break;
139        }
140      }
141
142      if(ctxElt == null){
143        ctxElt = Diva_contextFactory.eINSTANCE.createContextElement();
144        ctxElt.setName(name);
145        context.getElement().add(ctxElt);
146      }
147
148      return ctxElt;
149    }
150 }
```

Appendix B

Benchmarks

Contents

B.1 Comparative Study of SmartAdapters V1 and V2

Config.	#Aspects	#Join points	Aspect names	V1 (s)	V2 (ms)
Before Crisis	3	4	Normal, DefaultNotificationStrategy, EnergySaving	48.5	296
Alarm is raised	6	16	FireDepartment, EmergencyNotificationStrategy, FireStrategy, FirePlanner, TacticalPicturePlanner, Critical	194.2	594
Crisis confirmed	8	26	Hospital, FireDepartment, EmergencyNotificationStrategy, FireStrategy, FirePlanner, TacticalPicturePlanner, MedicalPlanner, Critical	418.9	797
Fire spreads	9	32	Hospital, Minestry, FireDepartment, EmergencyNotificationStrategy, FireStrategy, FirePlanner, TacticalPicturePlanner, MedicalPlanner, PushBroadcast	568.4	875
Victims evacuated	6	16	FireDepartment, EmergencyNotificationStrategy, FireStrategy, FirePlanner, TacticalPicturePlanner, Critical	182.4	609

<div align="center">Table B.1: Performances of SmartAdapters V1 and V2</div>

B.2   Comparison and Reconfiguration times in EnTiMid

-	Init	Night	Day	Day + Nurse	Day	Night	Day	Day + Nurse	Day
Comparison (ms)	385	420	418	463	432	419	431	473	425
Reconfiguration (ms)	5749	378	75	81	14	124	43	51	13

Table B.2: Comparison and Reconfiguration Times

Abstract

Society's increasing dependence on software-intensive systems is driving the need for dependable, robust, continuously available adaptive systems. Such systems often propose many variability dimensions with many possible variants, leading to an explosion of the number of configurations that is impossible to fully specify and validate at design-time because of time and resource constraints.

This thesis presents a Model-Driven and Aspect-Oriented approach to tame the complexity of Dynamically Adaptive Systems (DAS). At design-time, we capture the different facets of a DAS (variability, environment/context, reasoning and architecture) using dedicated metamodels. Each feature of the variability model describing a DAS is refined into an aspect model. We leverage these design models at runtime to drive the dynamic adaptation process. Both the running system and its execution context are abstracted as models. Depending on the current context (model) a reasoner interprets the reasoning model to determine a well fitted selection of features. We then use Aspect-Oriented Modeling techniques to automatically compose the aspect models (associated to the selected features) together in order to automatically derive the corresponding architecture. This way, there is no need to specify the whole set of possible configurations at design-time: each configuration is automatically built when needed. We finally rely on model comparison to fully automate the reconfiguration process in order to actually adapt the running system, with no need to write low-level reconfiguration scripts. An important point is that models embedded at runtime are really mirrors of what really happens in the running system. It is however possible to work on copies of these models, independently of the running system and resynchronize these copies with the reality to actually adapt the running system. In other words, our approach makes it possible to perform offline activities such as continuous design or prediction, while the system is running, but independently from it.

Résumé

La dépendance croissante de la société à l'égard des systèmes logiciels nécessite de concevoir des logiciels robustes, adaptatifs et disponibles sans interruption. De tels systèmes proposent souvent de nombreux points de variation avec de nombreuses variantes, conduisant à une explosion combinatoire du nombre des configurations. Il devient rapidement impossible de spécifier et de valider toutes ces configurations lors de la conception d'un système adaptatif complexe.

Cette thèse présente une approche dirigée par les modèles et basée sur la modélisation par aspects pour contenir la complexité de systèmes logiciels adaptatifs (Dynamically Adaptive Systems, DAS). Lors de la conception, les différentes facettes d'un DAS (variabilité, environnement/contexte, raisonnement et architecture) sont capturées à l'aide de différents méta-modèles dédiés. En particuliers, les variants de chaque point de variation sont raffinés à l'aide d'aspect (niveau model). Ces modèles sont embarqués à l'exécution pour contrôler et automatiser le mécanisme de reconfiguration dynamique. Le système courant et son contexte d'exécution sont abstraits par des modèles. Selon le contexte courant (modèle) un composant de raisonnement interprète le modèle de raisonnement et détermine un ensemble de variantes bien adaptées au contexte. Nous utilisons un tisseur d'aspects (niveau model) pour dériver automatiquement l'architecture correspondante à cette sélection de variantes. Ainsi, les concepteurs du DAS n'ont pas besoin de spécifier toutes les configurations : au contraire, chaque configuration est automatiquement construite lorsqu'il y en a besoin. Nous utilisons finalement une comparaison de modèle pour automatiser entièrement le processus de reconfiguration dynamique, sans avoir besoin d'écrire des scripts de reconfiguration de bas niveau. Les modèles embarqués à l'exécution sont des miroirs reflétant ce qui se produit vraiment dans le système courant. Il est cependant possible de travailler sur des copies de ces modèles, indépendamment du système courant et de re-synchroniser ces copies avec la réalité pour adapter réellement le système courant. En d'autres termes, notre approche permet d'exécuter des activités *offline* (pendant que le système fonctionne, mais indépendamment de lui) telles que la conception continue (continuous design) ou la prévision.