A development process for building adaptative software architectures

Ngoc Tho Huynh

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UNIVERSITE BRETAGNE LOIRE

THÈSE / IMT Atlantique

sous le sceau de l'Université Bretagne Loire

pour obtenir le grade de

DOCTEUR D'IMT Atlantique

Spécialité : Informatique

École Doctorale Mathématiques et STIC

Présentée par

Ngoc Tho Huynh

Préparée dans le département Informatique

Laboratoire Irisa

A Development Process for Building Adaptative Software Architectures

Thèse soutenue le 30 novembre 2017

devant le jury composé de :

Jean-Philippe Babau
Professeur, Université de Bretagne Occidentale / président

Sophie Chabridon
Directeur d'études (HDR), Télécom SudParis / rapporteur

Tewfik Ziadi
Maître de conférences (HDR), Université Pierre et Marie Curie / rapporteur

Christelle Urtado
Maître de conférences (HDR), Mines d'Alès / examinateur

Jérémy Buisson
Maître de conférences, Ecole de Saint-Cyr Coëtquidan / examinateur

Antoine Beugnard
Professeur, IMT Atlantique / directeur de thèse

Maria-Teresa Segarra
Maître de conférences, IMT Atlantique / invitée

Acknowledgements

I would like to profoundly acknowledge the people who have helped me during my PhD study:

First of all, I would like to sincerely thank my two supervisors, Antoine Beugnard, and Maria-Teresa Segarra, for guiding me at every step throughout my Ph.D. Before working with them, I knew nothing about research. Thanks to their guidance, and support, I have learned almost everything about research.

I would like to thank Prof. Sophie Chabridon and Prof. Tewfik Ziadi for spending time to review my thesis. Their comments and suggestions helped me to significantly improve my thesis. I would also like to thank my other thesis committee members, Prof. Jean-Philippe Babau, Prof. Christelle Urtado, and Prof. Jérémy Buisson for examining my thesis.

Many thanks to all members of the Processes for Adaptive Software Systems group in Département Informatique, for interesting discussions. Thanks to everyone in the department, particularly to Armelle Lannuzel who helped me on administrative work.

I gratefully thank my friends for their encouragement and support, and for sharing every time here. Particularly, thanks to An Phung-Khac whose encouragement had significant influence on this thesis.

Finally, I want to thank my parents, my wife, and everyone in my big family. They have always supported and encouraged me in every moment of my life.

Abstract

Adaptive software is a class of software which is able to modify its own internal structure and hence its behavior at runtime in response to changes in its operating environment. Adaptive software development has been an emerging research area of software engineering in the last decade. Many existing approaches use techniques issued from software product lines (SPLs) to develop adaptive software architectures. They propose tools, frameworks or languages to build adaptive software architectures but do not guide developers on the process of using them. Moreover, they suppose that all elements in the SPL specified are available in the architecture for adaptation. Therefore, the adaptive software architecture may embed unnecessary elements (components that will never be used) thus limiting the possible deployment targets. On the other hand, the components replacement at runtime remains a complex task since it must ensure the validity of the new version, in addition to preserving the correct completion of ongoing activities.

To cope with these issues, this thesis proposes an adaptive software development process where tasks, roles, and associate artifacts are explicit. The process aims at specifying the necessary information for building adaptive software architectures. The result of such process is an adaptive software architecture that only contains necessary elements for adaptation. On the other hand, an adaptation mechanism is proposed based on transactions management for ensuring consistent dynamic adaptation. Such adaptation must guarantee the system state and ensure the correct completion of ongoing transactions. In particular, transactional dependencies are specified at design time in the variability model. Then, based on such dependencies, components in the architecture include the necessary mechanisms to manage transactions at runtime consistently.

Keywords: Variability Modeling, Software Architecture, Consistent Dynamic Adaptation, Transactional Dependency, Transaction Management, Software Product Line.

Résumé

Les logiciels adaptatifs sont une classe de logiciels qui peuvent modifier leur structure et comportement à l'exécution afin de s'adapter à des nouveaux contextes d'exécution. Le développement de logiciels adaptatifs a été un domaine de recherche très actif les dix dernières années. Plusieurs approches utilisent des techniques issues des lignes des produits afin de développer de tels logiciels. Ils proposent des outils, des frameworks, ou des langages pour construire des architectures logicielles adaptatives, mais ne guident pas les ingénieurs dans leur utilisation. De plus, ils supposent que tous les éléments spécifiés à la conception sont disponibles dans l'architecture pour l'adaptation, même s'ils ne seront jamais utilisés. Ces éléments inutiles peuvent être une cause de soucis lors du déploiement sur une cible dont l'espace mémoire est très contraint par exemple. Par ailleurs, le remplacement de composants à l'exécution reste une tâche complexe, elle doit assurer non seulement la validité de la nouvelle version, mais aussi préserver la terminaison correcte des transactions en cours.

Pour faire face à ces problèmes, cette thèse propose un processus de développement de logiciels adaptatifs où les tâches, les rôles, et les artefacts associés sont explicites. En particulier, le processus vise la spécification d'informations nécessaires pour construire des architectures logicielles adaptatives. Le résultat d'un tel processus est une architecture logicielle adaptative qui contient seulement des éléments utiles pour l'adaptation. De plus, un mécanisme d'adaptation est proposé basé sur la gestion de transactions pour assurer une adaptation dynamique cohérente. Elle assure la terminaison correcte des transactions en cours. Nous proposons pour cela la notion de dépendance transactionnelle : dépendance entre des actions réalisées par des composants différents. Nous proposons la spécification de ces dépendances dans le modèle de variabilité, et de l'exploiter pour décider des fonctions de contrôle dans les composants de l'architecture, des fonctions qui assurent une adaptation cohérente à l'exécution.

Mots-clés : Modélisation de Variabilité, Architecture Logicielle, Adaptation Dynamique Cohérente, Dépendance Transactionnelle, Gestion de Transaction, Ligne de Produit.

Contents

Résumé en français

Sommaire

1 Introduction

L'adaptation d'un logiciel est la capacité d'un logiciel à s'adapter à son environnement d'exécution. Une adaptation est dite « dynamique » si le processus d'adaptation se déroule à l'exécution sans arrêter le système [Grondin 08]. Dans certains cas tels qu'un système médical ou un système de communication, un arrêt complet du système peut causer un dommage grave. Il est donc important de pouvoir développer de tels systèmes adaptatifs qui peuvent se modifier eux-mêmes.

Pour développer des logiciels adaptatifs, plusieurs travaux utilisent des techniques issues des lignes de produits (LdP). Ils utilisent des modèles pour spécifier la variabilité et l'architecture d'une ligne de produits tels que le modèle de variabilité. Des éléments communs (*commonality*) sont présents dans tous les produits et des éléments variables (variabilité) représentent des différences entre les produits [Bosch 01, Capilla 13]. La différence est spécifiée par des variantes dans

le modèle de variabilité. La correspondance (*mapping*) entre l'architecture de la ligne de produits et le modèle de variabilité est spécifiée. La description de la variabilité à la conception nous permet de configurer plusieurs produits différents. À partir des choix des éléments dans le modèle de variabilité (configuration de la ligne), une configuration de l'architecture est déduite. Les éléments choisis sont présents et activés dans le produit. Pour les besoins de l'adaptation, tous les éléments qui ne sont pas choisis dans la configuration sont disponibles dans le produit, même s'ils ne seront jamais utilisés. Ces éléments inutiles peuvent être une cause de soucis dans le déploiement sur une cible dont l'espace mémoire est très contraint par exemple. Enfin, les approches existantes ne fournissent pas d'instructions aux développeurs sur comment construire des architectures logicielles adaptatives.

Au-delà de la conception, à l'exécution, une nouvelle configuration peut être décidée et une nouvelle architecture correspondante déduite. En calculant la différence entre la configuration actuelle et la nouvelle configuration, des actions de reconfiguration sont générées. Elles sont exécutées pour modifier l'architecture logicielle actuelle. Cependant, la détermination du meilleur moment pour remplacer des composants dans le processus d'adaptation est une tâche complexe. Un processus d'adaptation assure non seulement la validité de la nouvelle configuration, mais doit aussi garantir la terminaison correcte des transactions en cours. Une telle adaptation s'appelle *l'adaptation dynamique cohérente*. Certains travaux proposent des solutions pour faire face à ce problème [Kramer 90, Ghafari 12a]. Tous sont basés sur la notion de « *transaction* ». Une transaction est une séquence d'actions exécutées par un ou plusieurs composants qui se termine en un temps borné. Lorsque le système est en train d'exécuter une transaction, aucun composant engagé dans cette transaction ne peut être remplacé. La terminaison d'une transaction peut dépendre de celle d'autres transactions. Cependant, ces travaux n'abordent pas la dépendance transactionnelle entre des composants dans le système et comment trouver le meilleur moment pour remplacer les composants.

Pour faire face aux problèmes cités ci-dessus, nous travaillons sur un processus de développement pour aider les développeurs à spécifier les informations nécessaires pour générer une architecture logicielle adaptative qui ne contient pas des éléments inutiles. Dans ce processus, des tâches, des rôles, des artefacts associés sont explicites. De plus, nous proposons une approche pour spécifier la dépendance transactionnelle. À partir du langage commun de variabilité (CVL) [OMG 12] et de son modèle de variabilité, nous introduisons une nouvelle contrainte, ***dependsOn***. Elle indique la dépendance transactionnelle entre composants. Nous proposons un mécanisme d'adaptation qui utilise les informations spécifiées à la conception pour développer des éléments dans le système, et les exploiter à l'exécution pour assurer une adaptation cohérente et sûre.

Le reste de ce document est organisé comme suit. La section 2 introduit le processus de développement proposé dans notre approche. La section 3 présente les mécanismes d'adaptation qui utilisent la spécification de dépendance transactionnelle pour assurer une adaptation dynamique cohérente. La section 4 décrit un prototype dédié au processus. La section 5 résume des travaux connexes et enfin, la section 6 conclut ce résumé et propose quelques perspectives.

2 Processus de développement des architectures logicielles adaptatives

Pour aider les ingénieurs à construire des logiciels adaptatifs, nous proposons un processus de développement qui comprend la spécification de la variabilité comme une première étape. Notre processus est basé sur CVL [OMG 12] pour spécifier la variabilité. Dans CVL, un modèle de variabilité se compose (1) d'*un arbre de VSpecs* qui est utilisé pour spécifier la variabilité et les parties communes d'une ligne de produits, et (2) de *points de variation* pour faire le lien entre l'arbre de *VSpec* et un modèle dit *de base* qui spécifie l'architecture de la ligne de produits. Dans CVL, un modèle *de résolution* est spécifié pour configurer le modèle de variabilité.

La figure 1 présente notre processus de développement d'architecture logicielle adaptative qui se compose de plusieurs étapes différentes. Notre processus est basé sur l'ingénierie de LdP qui distingue deux phases : l'ingénierie de domaine et l'ingénierie d'application.

2.1 L'ingénierie de domaine

Le haut de la figure 1 présente l'ingénierie de domaine. Il prend en charge de définir la variabilité, la *commonalité*, et l'architecture d'une ligne de produits. Le modèle de variabilité est spécifié en utilisant CVL, alors que le modèle de base est représenté en utilisant un langage de description d'architecture comme ACME [Garlan 00].

Les rôles des participants

Nous définissons trois rôles des participants dans cette phase : l'expert de domaine, l'ingénieur de domaine, et l'architecte de domaine. Un expert a une très bonne connaissance dans un domaine précis. Il fournit l'information nécessaire à l'ingénieur de domaine pour identifier la variabilité. L'ingénieur collecte l'information de domaine et construit le modèle de variabilité. L'architecte de domaine

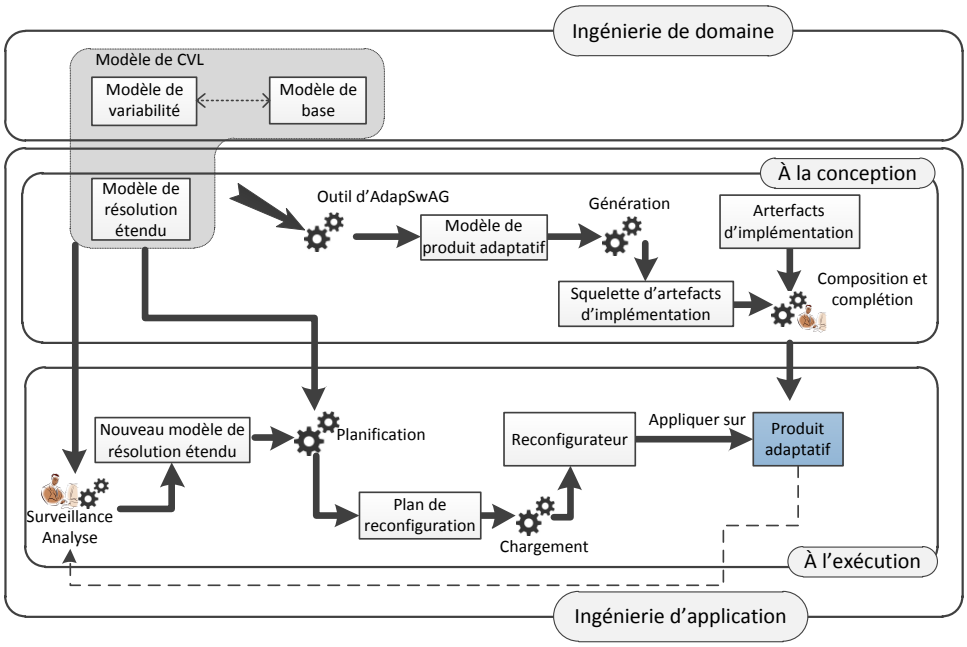


FIGURE 1 – Processus de développement de logiciel adaptatif

prend en charge la construction du modèle de base.

Un processus de spécification

La figure 2 présente une stratégie de spécification des modèles de variabi-

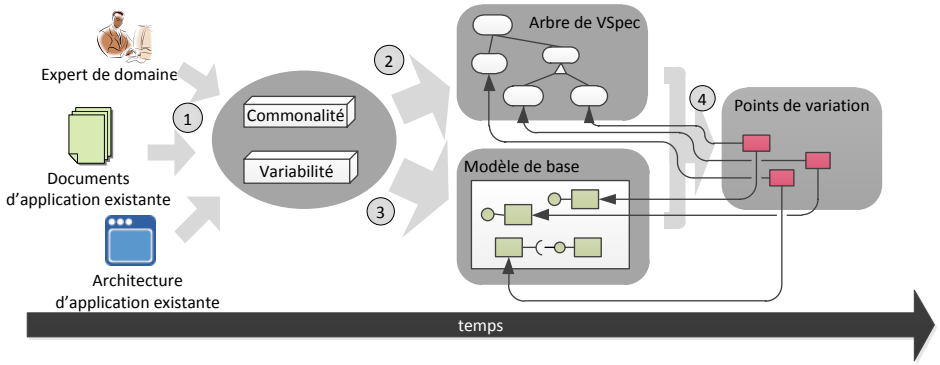


FIGURE 2 – Un processus de spécification des modèles

lité et de base. Cette stratégie se compose de quatre activités : 1) La première activité vise à identifier un ensemble de variabilité et commonalité dans le domaine ; 2) basé sur cet ensemble, un ingénieur de domaine construit un arbre de *VSpecs* ; 3) Une architecte logiciel définit le modèle de base ; 4) L'ingénieur de domaine fait le *mapping* entre l'arbre de *VSpecs* et le modèle de base.

La première activité est considérée comme un pré-requis. La quatrième activité ne peut être réalisée qu'après la deuxième et la troisième activité.

2.2 L'ingénierie d'application

La partie en bas de la figure 1 présente l'ingénierie d'application qui combine les artefacts obtenus dans l'ingénierie de domaine pour générer le produit adaptatif. Cette phase est divisée en deux sous-processus : celui de la conception et celui à l'exécution. En fonction des processus, les rôles sont définis.

Les rôles des participants

L'ingénierie d'application se focalise sur des activités pour générer le produit adaptatif et contrôler l'adaptation à l'exécution. Des utilisateurs fournissent des besoins pour qu'un concepteur de produit adaptatif crée un modèle de résolution. Le concepteur de produit adaptatif utilise un outil implémenté dans notre approche pour générer le modèle de produit adaptatif, puis le code exécutable. Des développeurs complètent le code généré, et l'empaquettent. Enfin, un ingénieur d'adaptation prend en charge de développer une unité de contrôle d'adaptation.

À la conception

En général, le sous-processus de conception vise à générer un produit adaptatif qui contient seulement les éléments nécessaires à l'adaptation. Pour faire cela, le sous-processus commence par la spécification d'un modèle de résolution. Aujourd'hui, cette spécification ne permet pas de distinguer les éléments nécessaires à l'adaptation de ceux rejetés. Par défaut donc, elle oblige à embarquer l'ensemble des éléments dans le produit. Nous proposons une extension pour améliorer la spécification.

La figure 3 présente un méta-modèle de CVL. La partie droite représente un extrait du modèle de résolution. Dans ce modèle, l'attribut de *decision* est utilisé pour sélectionner les éléments choisis. Un élément choisi par l'attribut *decision* va être présent et activé dans le produit. Si cet attribut est important, il ne suffit pas pour construire une architecture logicielle adaptative. En effet,

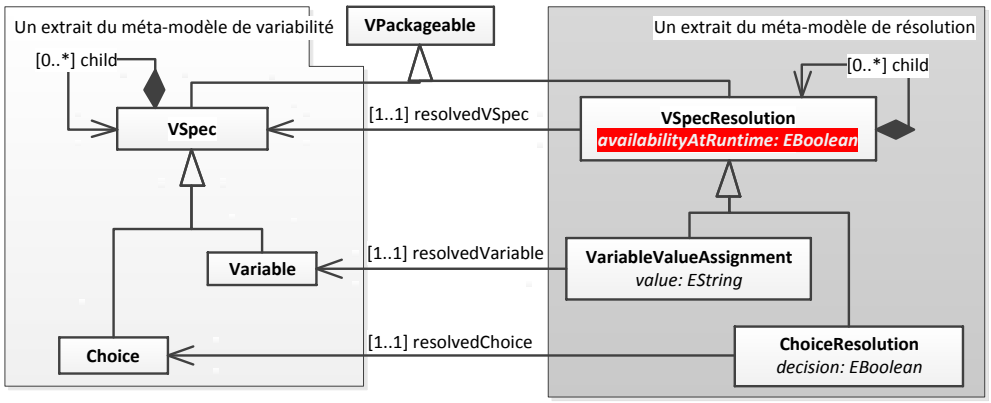


FIGURE 3 – Un extrait du méta-modèle de CVL

une telle architecture peut contenir des éléments qui ne sont pas activés dans le produit, tout en étant disponibles dans le produit pour un usage ultérieur.

Afin de spécifier de tels éléments, nous proposons un nouvel attribut, ***availabilityAtRuntime*** (à droite dans la figure 3). Il indique si l'élément correspondant est disponible ou pas dans le produit. Grâce à cet attribut, les éléments utiles ou inutiles à l'adaptation sont séparés.

Enfin, nous avons réalisé un outil AdapSwAG pour générer le modèle de produit adaptatif basé sur le modèle de résolution ainsi étendu. Il est utilisé pour générer un squelette d'artefacts d'implémentation. Il est combiné avec les artefacts d'implémentation existants ou complété par les développeurs pour créer le produit adaptatif.

À l'exécution

Lorsqu'une décision de changement se fait, un nouveau modèle de résolution (nouvelle configuration) doit être choisi. Il est comparé au modèle de résolution actuel pour trouver la différence avec l'architecture courante. À partir de cette différence, des actions de reconfiguration peuvent être générées dans un plan de reconfiguration. Le plan est envoyé à un reconfigurateur pour réaliser l'adaptation. Le moment où remplacer des composants est important pour assurer la cohérence de système. La section suivante va aborder ce problème.

3 Processus d'adaptation dynamique cohérente

Afin de réaliser une adaptation dynamique cohérente, dans notre approche, nous nous basons sur la gestion de transactions. Dans cette section, nous représentons comment gérer et exploiter la dépendance transactionnelle lors de la conception et à l'exécution.

3.1 Gestion de transactions

Un exemple

Nous étudions un exemple représenté dans la figure 4. Dans la figure, nous supposons un scénario d'adaptation pour le remplacement d'implémentation des composants, `Compression` et `Decompression`. Si ce remplacement se fait à l'exécution, il faut assurer que tous les messages qui ont été compressés par `Compression` soient décompressés par `Decompression` en utilisant le même algorithme avant de remplacer ces composants. Ceci indique qu'il y a une dépendance entre deux composants qui s'appellent *le composant démarreur* et *le composant terminateur*, respectivement. Dans ce document, nous appelons une telle dépendance *la dépendance transactionnelle*.

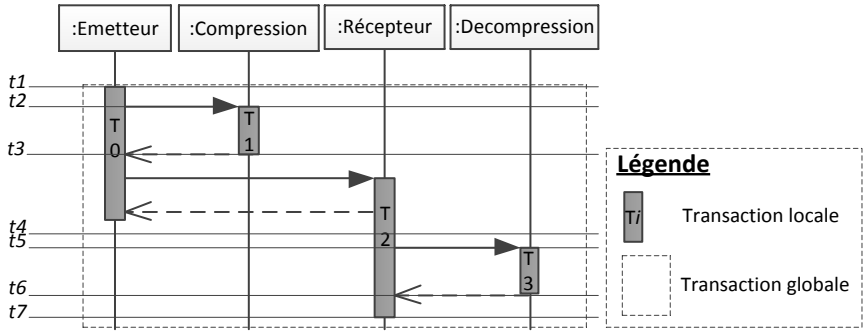


FIGURE 4 – Diagramme de séquence de l'exemple

La dépendance transactionnelle est définie sur la notion de transaction. Nous définissons deux types de transaction. Une transaction locale est définie comme une séquence d'actions exécutées par un composant qui se termine dans un temps borné. Une transaction globale est un ensemble des transactions locales exécutées sur des composants différents.

Dans l'exemple, avant de réaliser l'adaptation, la transaction globale doit

finir dans les composants dépendants (démarreur et terminateur). Ceci signifie que si la transaction `T1` a lieu, la transaction `T3` doit finir avant de pouvoir remplacer les composants. Du coup, il y a une dépendance transactionnelle entre `T1` et `T3`.

Spécification de dépendance transactionnelle

Afin de spécifier la dépendance transactionnelle, nous proposons une nouvelle relation, ***dependsOn***, ajoutée dans le modèle de variabilité. La figure 5 décrit un modèle de variabilité de l'exemple avec la relation ***dependsOn***. Dans la figure, les deux implémentations des fonctionnalités de compression et décompression sont considérées avec deux algorithmes, Run-Length Encoding (RLE) [Watson 03], et Lempel-Ziv (LZ) [Farach 98].

La relation indique que des messages qui sont traités par `RLECompression` doivent être traités par `RLEDecompression` avant de pouvoir réaliser l'adaptation. En termes de la transaction, si une transaction locale a lieu dans `RLECompression`, une transaction locale correspondante doit terminer dans `RLEDecompression`.

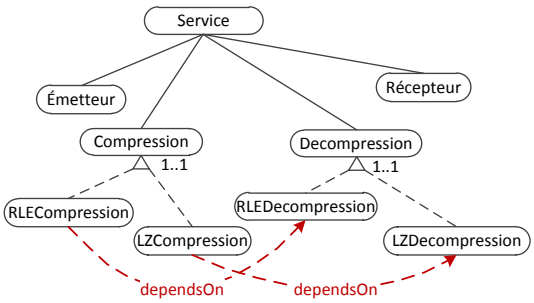


FIGURE 5 – Le modèle de variabilité avec la contrainte « ***dependsOn*** »

Gestionnaire de transactions et modèle de composant

Afin de gérer les transactions dans les composants dépendants, un gestionnaire de transactions doit être ajouté dans l'architecture. Il doit communiquer avec les composants dépendants dans l'architecture et recevoir des informations du début de la transaction de `RLECompression` ou `LZCompression`, et celles de la fin de la transaction de `RLEDecompression` ou `LZDecompression`. Grâce à ces informations, le gestionnaire peut identifier quand une transaction globale finit dans les composants dépendants. En communiquant avec le gestionnaire de transactions, le reconfigurateur peut calculer le meilleur moment pour remplacer des composants.

Pour fournir l'information de contrôle transactionnel au gestionnaire de transaction, les composants dépendants ont besoin d'avoir la capacité de communiquer avec lui. En outre, ils peuvent recevoir et réaliser des actions de reconfiguration déclenchées par le reconfigurateur. De plus, l'adaptation dans notre approche est basée sur l'isolement des composants [1] qui sont remplacés. Donc, nous proposons un nouveau modèle de composant (comme décrit dans la figure 6) qui consiste en quatre parties : un calcul (le cœur fonctionnel), une connexion, une barrière, et un contrôle. Les parties de calcul et de connexion sont les parties principales du modèle de composant. La barrière est utilisée pour isoler le composant du reste du système pour l'adaptation. Le contrôle prend en charge la réalisation des actions de reconfiguration reçues du reconfigurateur et l'envoi des informations de contrôle transactionnel au gestionnaire de transactions.

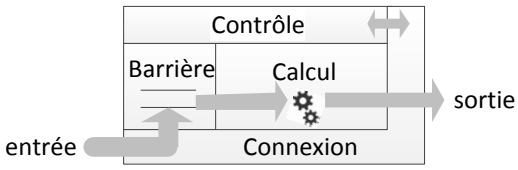


FIGURE 6 – Le modèle de composant

Grâce au modèle de composant et à la spécification de dépendance transactionnelle, les composants dépendants sont conçus avec des fonctionnalités convenables pour assurer la cohérence de l'adaptation.

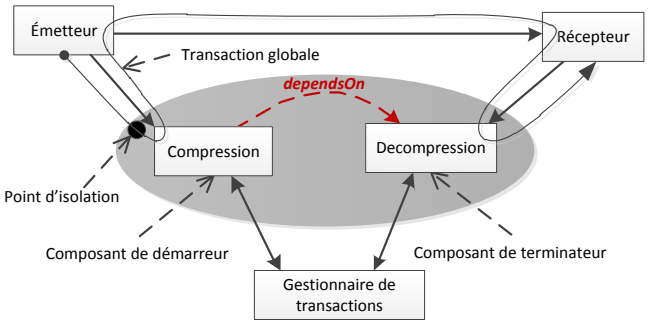


FIGURE 7 – Groupe de composant remplacé

La figure 7 décrit un exemple où un groupe des composants, `Compression` et `Decompression`, va être remplacé. Basé sur la spécification de dépendance transactionnelle, les composants savent qu'ils doivent informer le gestionnaire

1. Basé sur l'idée dans le domaine de la chirurgie, un chirurgien doit isoler l'organe à remplacer.

de transactions du début et de la fin des transactions dans le groupe. Le gestionnaire de transactions participe alors au processus d'adaptation et fournit au reconfigurateur les informations sur les transactions dans les composants dépendants.

La section suivante présente une stratégie d'adaptation réalisée dans notre approche.

3.2 Stratégie d'adaptation

Une stratégie d'adaptation utilisée dans notre approche est basée sur l'isolement du groupe des composants (le groupe de placement) qui va être remplacé par un autre (le groupe de remplacement). En fait, lors de la réception d'une commande d'isolement du reconfigurateur, les composants de frontière du groupe de placement vont activer leur barrière. Toutefois, afin de terminer des transactions en cours dans le groupe, la barrière permet aux messages qui viennent de l'intérieur du groupe de passer. De nouvelles requêtes qui viennent de l'extérieur du groupe vont, elles, être bloquées à la barrière. Cependant, afin de terminer les transactions dans les composants dépendants, la barrière permet aux messages traités par le composant démarreur de passer la barrière du composant terminateur.

4 Prototype

Nous avons développé un outil qui automatise certaines tâches du processus de développement. En particulier, il aide l'ingénieur à générer le produit adaptatif. En outre, cet outil contient un module qui permet de valider des configurations de la ligne de produits spécifiées dans des modèles de résolution.

De plus, afin de vérifier notre approche, nous avons implémenté l'exemple mentionné dans la section précédente. Nous suivons les étapes proposées dans la section 2 pour construire le produit. Les composants dans le produit adaptatif sont générés sur le modèle de composant iPOJO [Escoffier 07] et sont conformes à notre modèle de composant pour la gestion des transactions et de l'adaptation.

5 Travaux connexes

Plusieurs approches sont basées sur la ligne de produits pour construire le produit adaptatif. Ils utilisent des modèles différents pour spécifier la variabi-

lité tels que les approches dans [Lee 06, Cetina 13] qui utilisent le modèle de fonctionnalité proposé par Kang *et al.* [Kang 90], [Morin 09a, Bencomo 08] avec le modèle de variabilité orthogonal proposé par [Pohl 05], ou [Pascual 14] avec CVL. Ces modèles sont configurés pour construire le produit adaptatif. Cependant, ils ne distinguent pas les éléments disponibles pour l'adaptation. Par défaut, tous les éléments sont disponibles pour l'adaptation. L'approche dans [Cetina 08a] élimine quelques éléments dans le modèle de variabilité. Toutefois, elle est basée sur un ensemble des scénarios d'adaptation différents prévus à la conception.

Aucune de ces approches n'offrent d'instructions (processus) à l'ingénieur pour concevoir des modèles. Notre approche explicite les étapes du processus pour générer une architecture logicielle adaptative. L'architecture générée contient seulement les éléments nécessaires pour l'adaptation.

Pour identifier le meilleur moment pour remplacer des composants, la plupart des approches sont basées sur la gestion de transactions et définissent l'état « sûr », tels que « quiescent » dans [Kramer 90], ou « tranquil » dans [Vandewoude 07]. Cependant, ces états sont considérés sur un *unique* composant sans dépendance transactionnelle. L'approche proposée par Ghafari *et al.* dans [Ghafari 15] considère un groupe de composants. Ils définissent un contrôleur de transactions pour initier et terminer des transactions. Grâce au contrôleur, le moment convenable est explicite. Toutefois, c'est une approche « ad-hoc » . Ils ne spécifient pas la dépendance transactionnelle à la conception et ne l'exploitent pas pour gérer des transactions pour l'adaptation. Notre approche utilise une relation ajoutée dans le modèle de variabilité pour spécifier la dépendance transactionnelle. Cette relation est nécessaire pour identifier le meilleur moment pour le remplacement des composants.

6 Conclusion

Contributions

Dans cette thèse nous avons proposé un processus de développement d'architecture logicielle adaptative et un mécanisme d'adaptation basé sur la spécification à la conception de dépendances transactionnelles pour assurer une adaptation dynamique cohérente. En fonction de ceci, nos contributions principales sont déterminées comme suit :

1. *Un processus de développement basé sur la modélisation de la variabilité pour construire des architectures logicielles adaptatives qui ne contiennent pas d'éléments inutiles pour l'adaptation* : Ce processus vise à aider les

développeurs à construire des architectures logicielles adaptatives qui comprennent uniquement des composants potentiellement utilisés. Une des activités importantes de ce processus est la spécification de logiciel basée sur les modèles. Dans ce processus, nous proposons l'utilisation du modèle de CVL que nous avons étendu pour que le produit résultant ne comprenne pas d'éléments qui ne seront jamais utilisés dans l'architecture adaptative.

2. *Un mécanisme d'adaptation basé sur la spécification de la dépendance transactionnelle pour assurer l'adaptation dynamique cohérente* : Ce mécanisme est considéré à la conception et à l'exécution. À la conception, le méta-modèle de CVL est étendu pour spécifier la dépendance transactionnelle. La relation ajoutée est exploitée à l'exécution pour trouver le meilleur moment pour l'adaptation. D'autre part, elle est également utilisée pour générer des actions de reconfiguration dans le plan de reconfiguration. En outre, un modèle de composant est proposé pour supporter la gestion de transactions et l'adaptation cohérente.

Perspectives

Notre approche ouvre plusieurs perspectives différentes. D'abord au niveau de la spécification, le modèle de variabilité ne change pas au fil de temps. Il pourrait être intéressant d'étudier les conséquences de *l'évolution* d'un modèle de variabilité. Ensuite, le processus d'adaptation est centralisé. On pourrait envisager un processus plus réparti. Enfin, une réelle application a besoin d'être étudiée avec l'approche proposée pour évaluer sa performance.

Chapter 1

Introduction

Contents

1.1 Problem Statement

Software adaptation is the capacity of a software to adapt to a changing operating environment. The operating environment includes anything around software execution such as user requirements, end-user input, external hardware devices and sensors, or program instrumentation [Oreizy 99]. Adaptation is said to be dynamic if the adaptation process is performed at runtime, i.e., without stopping the whole application while modifying it [Grondin 08]. In certain circumstances, dynamic adaptation is needed for software systems such as medical, finance, or telecommunication systems since stopping the software have consequences for business, or is even dangerous for humans. Hence, building autonomous software that is able to adapt itself is mandatory. Through such an adaptation, called *adaptation process*[1], the software system moves from the current version to a new one. For example, a message communication system can use encoding/decoding functions to encrypt messages during communication. Depending on the security level, the used encoding/decoding functions

1. In this dissertation, an adaptation process can be understood as an adaptation process at runtime.

may be different. To avoid interruption of the message communication system, encryption functions must be changed at runtime.

Over the past decade, software architecture has emerged as an important field in software engineering for representing the design of software systems [Hussein 11]. In software architectures, components represent the computational elements and data stores of the system, while connectors represent interactions among them via interfaces. This is the core architectural representation adopted by a number of architecture description languages (ADLs), such as ACME [Garlan 00] and FractalADL [Coupaye 07]. Architecture-based adaptive software development has emerged and is widely used [Cheng 02c, Floch 06, Cheng 09]. Adaptive software architectures are intended for systems whose configuration can be changed at runtime.

Through architecture-based adaptation, an adaptation process replaces the active variants by other through *reconfiguration actions*. The actions include start/stop, add/remove components and establish/destroy connections between them. Still, the replacement of one component by another one remains a complex task. This complexity is caused by the component state which is contained in the active components [Vandewoude 05] and their connections. Therefore, an adaptation process not only includes the actions for components replacement, but also take *state transfer* activities into account.

Software Product Line (SPL) paradigm has emerged as one of the approaches to develop software products or software-intensive systems at lower cost, in shorter time, and with higher quality [Pohl 05]. SPLs focus on managing and building multi software products from a set of previously developed and tested artifacts. A fundamental principle of SPL is variability management [Hallsteinsen 08] that encompasses the activities of explicitly representing variability in software artifacts throughout the lifecycle, managing dependencies among different variabilities, and supporting the instantiation of the identified variabilities [Schmid 04]. In SPLs, a desired product is built by deciding about selected and unselected elements at design time, and it contains all the necessary elements to respond to user requirements. However, it can not be modified at runtime.

Dynamic Software Product Lines (DSPLs) extend classic SPL engineering approaches by moving their change capabilities to runtime [Hinchey 12], i.e., a new product version can be derived at runtime by modifying the running product. In order to derive a new product at runtime, the elements of the product line must be *available at runtime* in the product. They do not participate on the calculation process of the initial product, they are *inactive elements*. Therefore, a product contains all the elements of the DSPL even if some of them are inactive and will not (or even cannot) be used for the target execution environment.

This can be a problem for limited deployment targets. Hence, identifying the inactive elements that will be used for adaptation is necessary to reduce the size of the adaptive product, increase security and reduce adaptation costs. For example, in a message communication system, only encryption algorithms that may be used in the product should be included in it, even if some of them are inactive in the initial product.

In order to realize adaptation processes, a control unit, called *reconfigurator*, needs to be added into the adaptive software architecture. This control unit can be manually created with embedded reconfiguration actions such as [Martinez 15c]. This increases development costs, and is error-prone. The reconfigurator may also be designed as an interpreter of reconfiguration plans. It can execute a received reconfiguration plan from an external element. Reconfiguration plans can be manually programmed or be automatically generated. The reconfiguration plan based on the manually programming increases development costs and is error-prone. Thus, *reconfiguration plans* should be generated automatically and be injected into the reconfigurator to realize adaptation process.

Determining the best moment to replace components is a difficult task. The adaptation process should ensure not only the validity of the resulting product but also should preserve the correct completion of ongoing activities and minimise disruption of the services. Such an issue has been addressed by Kramer and Magee [Kramer 90] and is based on determining a *safe status*[2] named *quiescence* in the paper. Identifying the safe status is based on the concept of "transaction". A transaction is a sequence of actions executed by one or several components that completes in bounded time. When the system is executing a transaction, no component involved in this transaction can be replaced. According to [Kramer 90], two transactions are dependent on each other if the completion of a transaction may depend on the completion of other one. In terms of architecture, requiring services of a component to another one is considered as *static dependency* or *architectural dependency*. The static dependency between components indicates that if a component exists, the other one must be present in the architecture. In terms of transaction, the completion of a transaction can require the completion of another transaction on the other component. Such a dependency is called *dynamic dependency* or *transactional dependency*. Quiescence-based approaches only consider *transactional dependencies* between components that are adjacent in the architecture where a component requires services of other one. In order to build adaptive software, we will show that based on the transactional dependency between adjacent components is not enough since dynamic dependencies between non-adjacent components should

2. In this thesis, we use *status* instead of *state* to differentiate the moment for adaptation and the state of variables and properties of components like to [Vandewoude 07].

be taken into account for finding the best moment for replacing components.

From the aforementioned problems, we argue that the adaptive software architecture development is a complex task and an appropriate development paradigm that deals with these difficulties is needed.

1.2 Challenges

Based on the problem statement, we identify five challenges for developing adaptive software architectures. The first two challenges are concerned with the development process at design time to generate an adaptive product. The others are related to exploiting information specified at design time for adaptation at runtime.

- *Modeling variability and commonality for adaptation (C1):* Developing adaptive software could be based on SPL engineering to specify the variability and commonality of system variants. Variability of functions should be clearly separated from the adaptive software architecture, and variation points need to be clearly identified. In addition, a process to guide engineers on how to specify them is necessary.

- *Configuring and automatically building adaptive architecture (C2):* The adaptive architecture should only contain elements that can be used in the product. Thus, information related to such elements should be explicitly specified for configuration. On the other hand, based on the information given at design time, the adaptive architecture should be automatically generated to ensure consistency from the abstract level to the concrete one in the development process.

- *Supporting state transfer (C3):* An adaptive software system must guarantee its state before and after adaptation, i.e., the state should be transferred from the running variant to the new one when executing adaptation process.

- *Automatically planning adaptation (C4):* In order to reduce the time and adaptation costs, reconfiguration plans should be automatically generated instead of being manually implemented.

- *Ensuring consistent dynamic adaptation (C5):* System consistency must be ensured after adaptation. That means that the adaptation process should preserve the correct completion of ongoing activities.

These challenges can be divided into two groups: one related with what should be considered for specifying and building automatically adaptive architecture at design time (C1, C2), and another one related with dynamic adaptation at runtime (C3, C4, and C5). The challenges are closely related to each other. Challenges C1 and C2 emphasize two different steps in the development process. Challenge C3 is concerned with C5 as one important activity for ensuring consistent dynamic adaptation is state transfer. Therefore, a solution for challenge C3 aims at providing the necessary information for this activity. On the other hand, challenge C4 is related with C1, and C3. Moreover, a solution for challenge C5 aims at finding the best moment to execute reconfiguration plan mentioned in challenge C4.

1.3 Research Methodology

This section presents the research methodology followed from the beginning to the final results. It is considered as a research process in which a sequence of activities that include explorations, choices, and re-orientations have been performed before identifying what would be the contributions. In a PhD, these research activities are linear but not smooth.

First of all, a research topic was determined. It originated from the research history and orientation of the PASS team [3]. One of the main research orientations of the group is to ease the design and development of adaptive software systems based on models. One of the models is variability model (or feature model) that is used in SPL engineering to specify the variability and commonality of a product line. Therefore, the first phase of our research process focused on using techniques of SPL engineering to build adaptive systems. From this aspect, our first research question (RQ) was as follows:

RQ1. *How to build adaptive software architectures?*

In order to answer this question, we have done a survey on the state of the art on modeling adaptive systems. Particularly, approaches based on variability modeling were analyzed. Approaches such as [Trinidad 07, Cetina 09a, Phung-khac 10] use feature models [4] to specify and manage variability and build adaptive software. The change points in the adaptive architecture are presented through variation points in the feature model. However, we saw that a variation

3. `http://recherche.telecom-bretagne.eu/pass/`

4. The terms of "feature model" or "variability model" signify the same sense, in this dissertation, we will use the terms according to authors of each approach.

point is considered as a feature in the feature model, and links with the adaptive architecture were omitted. From the studied approaches we concluded that separation between variation points and variability specification is necessary. After a review of the literature of existing variability modeling approaches such as Feature-Oriented Domain Analysis (FODA) [Kang 90], Orthogonal Variability Model (OVM) [Pohl 05], Common Variability Language(CVL) [OMG 12], etc, CVL approach has emerged as the best approach. It clearly separates between the variability specification, variation points, and software architecture. CVL offers models and tools but it does not offer methods to use them. Thus, a process should be proposed to guide engineers on how to specify these models, and build the adaptive software architecture. The results of this work were reported in [Huynh 16a].

We continued our research by studying existing approaches using CVL for building adaptive software. All of the approaches, e.g., [Cetina 09b, Pascual 14] assume that all elements are embedded in the final product and deployed at runtime, even if some of them will never be used. Therefore, we proposed to extend CVL meta-model to allow specification of unuseful elements for a particular final product. More detail on this aspect is found in Chapter 4.

Next, after a solution was proposed to build adaptive software architectures, a control unit needs to be added in to the adaptive architecture to realize adaptation processes. However, determining the appropriate moment for components replacement is necessary to ensure the system consistency. This aspect started as the second phase in our research process:

RQ2. *What is the appropriate moment to start components replacement in an adaptation process?*

To answer this question, we investigated a case study described in Chapter 5. Moreover, we analyzed literature on related approaches such as [Kramer 90, Vandewoude 07, Ma 11, Ghafari 12a] and applied them to it to find limitations of each approach. We saw that all these approaches are based on the concept of transaction to determine a *safe status* for adaptation. According to [Kramer 90, Vandewoude 07], this status involves a single component. We tried to apply them to our case study, and found that [Kramer 90, Vandewoude 07] can not ensure a reliable adaptation in a system in which replaced components depend on each other, e.g., encoding and decoding components. That means that computation errors arise in the system after adaptation.

In terms of transactions, we use the term transactional or dynamic dependency when transactions on a component depend on transactions on the other one. We tried to better understand such dynamic dependencies by studying ap-

proaches in [Ma 11, Ghafari 12a]. Unfortunately, these are ad-hoc approaches, i.e., there are no general solution to manage dynamic dependencies. They are not interested in specifying such dependencies for building adaptive software.

This analysis led us to conclude that a general solution is necessary to manage dynamic dependencies. This solution should include dynamic dependency specification. The specification is made at design for building adaptive architectures and exploited at runtime for generating adaptation actions. Part of the results from this study is presented in [Huynh 16b]. More detailed answers to these questions are found in Chapter 4 and Chapter 5, respectively.

Finally, the models used to specify the variability and the software architectures are based on the EMF framework integrated in Eclipse. Therefore, to validate our propositions, we implemented tools as Eclipse plug-ins supported for building adaptive software architectures from the models specified in the EMF framework. Those tools were applied on case studies to check their feasibility. Moreover, we reused the case study adapted from [Ma 11, Ghafari 15] to validate the aspect of transaction management in our approach. Although this case study is simple, it is not trivial. It has the necessary properties to cope with the issues of managing transactional dependency.

1.4 Contributions

Based on the challenges and research questions mentioned in the previous sections, a comparison with the related approaches is presented in Chapter 2. No existing approaches support all challenges and answer the research questions. Therefore, we worked on a development process that encompasses all the challenges for building adaptive software architectures. According to this, our main contributions are identified as follows:

1. **Development process based on variability modeling to build adaptive software architectures that do not contain the unnecessary elements for adaptation:** This development process aims at guiding developers building an adaptive software architecture that includes only components that will potentially be used. One of the important activities of this development process is software specification based on models. In this process, we propose to use the models and basic tools of CVL. Moreover, we have extended them so that the resulting adaptive product does not include elements that will never be used in the architecture.

2. **Adaptation mechanisms based on dynamic dependencies specification to ensure consistent dynamic adaptation:** This adaptation

mechanism is considered from design to runtime. At design time, the CVL meta-model is extended for specifying dynamic dependencies. Moreover, this specification is exploited at runtime to find the best moment for adaptation. On the other hand, it is also used for generating reconfiguration actions in the reconfiguration plan.

Both contributions are based on specifying the necessary information in models at design time. The former focuses on design time to build the adaptive software architecture that contains necessary elements for adaptation. In comparison with the related approaches, we identify just the necessary elements, whereas, the existing approaches suppose that all components at design time exist in the product at runtime or are added on-the-fly. The latter encompasses dynamic dependencies specification at design time and mechanisms for adaptation at runtime. Existing approaches are based on the concept of transaction, but, they are ad-hoc approaches. They do not take dynamic dependencies specification into account.

1.5 Structure of the Thesis

Figure 1.1 shows the structure of this thesis. It is structured in three main parts: state of the art, contribution, and epilogue. Part I provides a foundation to read Part II. Part III concludes the dissertation.

Part I consists of two chapters. *Chapter 2, State of the Art,* considers the state of art and offers the reader the foundation and concepts about software development process, CVL, sofware architecture and its adaptability. *Chapter 3, Related software adaptation approaches,* discusses the related approaches based on the challenges identified in this chapter. The conclusion of Chapter 3 determines limitations of existing approaches to position our contribution.

Part II consists of two chapters. *Chapter 4, Adaptive Software Architecture Development Process,* presents a development process based on variability modeling for building an adaptive software architecture. It focuses on coping with the first group of challenges. We extend the variability model from CVL and build a tool implemented as an Eclipse plug-in in order to automatically build architecture model. Additionally, *Chapter 5, Consistent Dynamic Adaptation Process,* focuses on adaptation mechanisms to ensure consistent dynamic adaptation. This chapter mentions the specification of the dynamic dependency and adaptation mechanism that uses the dynamic dependency specification to find the best moment for realizing adaptation. To support consistent dynamic adaptation, we have implemented a module to validate the new product configuration. Moreover, a prototype has been done to validate the propositions in

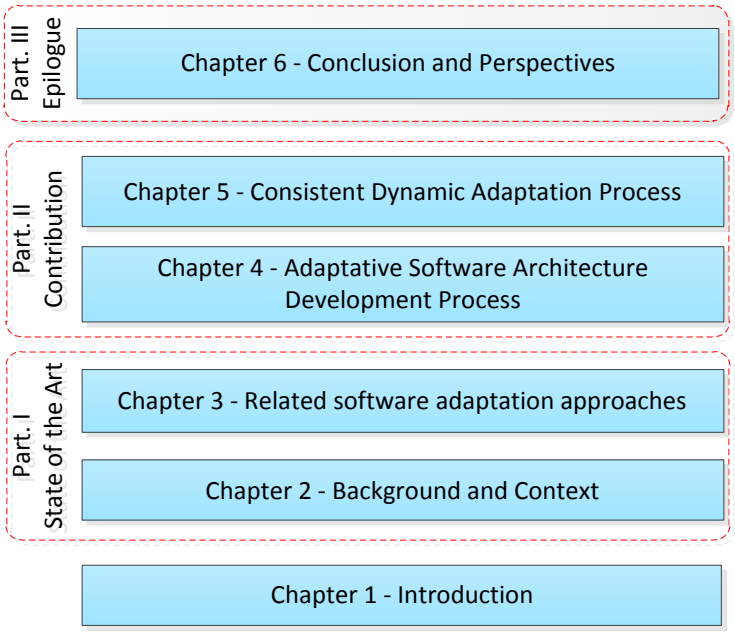


Figure 1.1 – Structure of the thesis

this chapter.

Part III contains *Chapter 6, Conclusion and Perspectives,* that highlights the contributions of the dissertation. In addition, some perspectives of future work are also anticipated.

Part I

State of the Art

Chapter 2

Background and Context

Contents

2.1 Chapter Overview

This chapter presents concepts and principles related to software development in which software product line engineering (SPLE) is highlighted. One of the main techniques in SPLE is variability modeling. Moreover, architecture based software development has arisen as an effective way to develop software systems that encompasses adaptive software development. The goal of this chapter is to introduce and define a base of knowledge that will be used throughout this dissertation.

Structure of the Chapter

Section 2.2 introduces approaches to software development including Model-Driven Engineering and SPLE. An important technique in SPLE is variability modeling. In our approach, the common variability language (CVL) used to specify the variability is presented in section 2.3.

Section 2.3 presents the CVL approach. A summary of CVL is presented to see a general view of this approach. In this approach, software architecture specification plays a main role for building a software architecture.

Section 2.4 shows concepts related to software architecture and elements used to fabricate software architectures. In this section, we will also present what an architecture description language is.

Section 2.5 presents terminologies in software adaptation. Then, we present some fundamental knowledges related to software adaptation. Finally, it presents needs to ensure consistent dynamic adaptation.

Finally, Section 2.6 summarizes the chapter.

2.2 Software Development

This section provides a brief introduction of two software development methodologies, including Model-Driven Engineering and Software Product Line Engineering (SPLE). The former presents the concepts related to Model-Driven Engineering, and focuses on Model-Driven Architecture and Domain Specific Modeling. In the latter, we highlight the role of subprocesses in SPLE, and techniques to manage variability and configure products.

2.2.1 Model-Driven Engineering

During the last decade a new trend of approaches used in the software engineering field has emerged, called Model-Driven Engineering (MDE). Their aim is to cope with the challenge of the increasing complexity and productivity in software development. MDE is based on using models as the critical artifacts in the software development. The aim of using models to reduce software complexity has been around for many years [Schmidt 06].

No existing definition for models is accepted by software engineering community, because, nobody can just define what a model is, and expect that other people will accept this definition [Ludewig 03]. Many model definitions are indicated in [Creff 13], for example, "*a model is a simplification of a system built with an intended goal in mind*" [Bézivin 01], or "*a model of a system is a description or specification of that system and its environment for some certain purpose*" [OMG 03], etc. But, according to the scope of this dissertation, we consider (a) model(s) as:

> *"An abstraction or high-level specifications, a description of a system*, or *some aspects of a system*" [Czarnecki 05a, Kühne 06, France 07, Filho 14].

"Models help in developing artifacts by providing information about the consequences of building those artifacts before that are actually made" [Ludewig 03]. The system that is specified by a model may or may not exist at the time when the model is created. Models are created to serve particular purposes, for example, presenting a human understandable description of some aspects of a system or information in a form that can be mechanically analyzed [France 07]. In terms of development process, models are used as the primary source for documenting, analyzing, designing, constructing, deploying and maintaining a system [Truyen 06].

The main activities in MDE focus on creating models to specify the systems [Kleppe 03]. In MDE, many different models can be created in which each kind can be specific to represent problems of a distinct domain of knowledge. Instead of using only one single kind of model to serve as unified language to represent any problems, Domain Specific Modeling Languages (DSML) are dedicated to particular areas, such as medical or avionics systems [Filho 14].

MDE technologies combine: 1) DSML where the application structure, behavior, and requirements of a type of systems are formalized within particular domains, such as software-defined radios, avionics mission computing, online financial services, warehouse management, or even the domain of middleware platforms; 2) transformation engines and generators that analyze some aspects

of models and then synthesize different types of artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations [Schmidt 06].

As mentioned in [France 07], there are major initiatives of MDE such as Object Management Group's (OMG) Model Driven Architecture [Kleppe 03], Microsoft Software Factory [Greenfield 04], Model Integrated Computing (MIC) [Sztipanovits 97], etc. In these initiatives, Model Driven Architecture (MDA) is considered as one of the best initiatives of MDE which is a registered trademark of OMG. On the other hand, Domain Specific Modeling (DSM) is also considered as another branch of MDE.

2.2.1.1 Model-Driven Architecture

MDA is an approach proposed by the OMG to build systems using models. The models allow to represent the systems independently of a specific platform. It takes full advantage of the basis principle of separation of concerns to separate the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform [OMG 03]. Both specifications are defined as models, called Platform Independent Models (PIMs), and Platform Specific Models (PSMs), respectively. PIMs specify the structure and functions of a system in a way that abstracts out technical details. Whereas, PSMs are created from PIMs by combining them with the necessary information to produce implementations for a selected platform. The transformation from PIMs to PSMs is performed by using model transformations. Code generators are used to generate the source code executable in the target platform.

In this dissertation, we consider the component-based architecture model as a PSM. It is used to generate implementations skeleton conforming the target platform by using a specific code generation module implemented in our approach.

2.2.1.2 Domain-Specific Modeling

Domain-Specific Modeling (DSM) is a software engineering methodology for designing and developing systems. It is based on specifying a solution using domain concepts or problem-level abstractions. DSM raises the level of abstraction and hides programing languages [Kelly 07].

DSM is usually described by using a Domain-Specific Modeling Language (DSML). DSML is a language to express problems of a particular domain. It

is defined as a set of meta-models used to describe structures, behaviors and relationships among the problem domain concepts including their semantics and associated constraints [Schmidt 06].

DSM also includes a domain specific code generator that uses the domain-specific models to fully generate artifacts of concrete applications. The artifacts may be text, executable source code, or immediate models. With DSM, the human's manual intervention in the artifacts is significantly reduced. Therefore, DSM improves the productivity, and product quality [Kelly 07].

In this thesis, we use models to specify software systems. The models are used to describe different aspects of a system, such as the variability model, the architecture model, etc.

2.2.2 Software Product Line Engineering

Software Product Line Engineering (SPLE) has been proposed as a methodology to develop similar software products and software-intensive systems at lower costs, in shorter time, and with higher quality [Clements 01, Pohl 05, Babar 10]. In terms of costs, as stated by [Pohl 05], SPLE offers benefits when producing products. Figure 2.1 shows the costs of the development of n different products. The solid line sketches the costs of producing a single product, whereas the dotted line presents the costs of developing products using SPLE. As described in the figure, before the break-event point, the cost of developing products in SPLE is relatively high. It is significantly reduced for larger quantities of products built with SPLE. The break-even point where the two lines intersect presents the same cost for developing the products separately as well as for developing them by product line engineering. Empirical investigations indicate that the break-even point is reached around three or four products [Weiss 99, Clements 01].

In the SPLE community, an Sofware Product Line (SPL) or software family is defined as a set of similar products built from re-usable artifacts. A desired product is configured and derived by reusing the available artifacts [Arboleda 13] such as common architectures, software components, models, documents, or other elements useful to develop systems. To enable reuse on a large scale, SPLE identifies and manages commonality, i.e., the common characteristics (or features) of all products in the family, and variability, i.e., characteristics (or features) that may be different in different products, across a set of system artifacts [Babar 10]. Variability is defined as the ability to change or customize a system [Gurp 01]. It plays a central role in an SPL development process and is also a concern in all stages of the life cycle of a product.

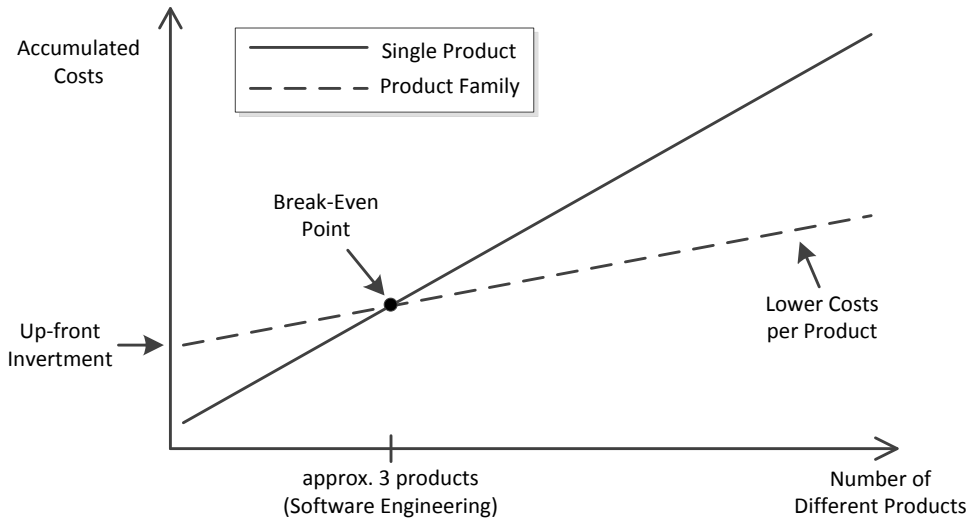


Figure 2.1 – Costs of software product development (adopted from [Pohl 05])

2.2.2.1 Software Product Line Process

To make reuse possible, the SPLE paradigm separates two processes, domain engineering and application engineering to build individual products [Czarnecki 00, Harsu 02]. The former can be considered as a "development for reuse", while the latter is a "development with reuse". Figure 2.2 shows those processes.

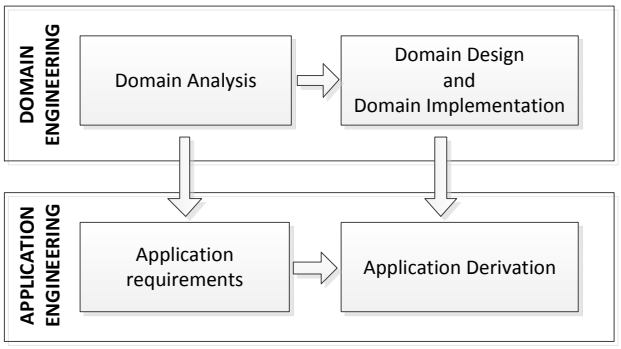


Figure 2.2 – Domain engineering and Application engineering

Domain Engineering

Domain engineering focuses on the analysis of the domain, the commonalities and the variabilities for reusable artifacts as requirements, design models, architectures, etc. Like traditional software engineering, the domain engineering includes analysis, design, and implementation activities. Domain analysis consists of determining the scope of a product line, identifying the common and variable features among the family members, and creating structural and behavioral specifications of a product line. Domain design aims at developing a common architecture for all the members of a product line, and a plan to build individual products based on the reusable artifacts [Czarnecki 05a]. Finally, domain implementation consists of implementing reusable artifacts as components, generators, and DSLs [Czarnecki 05a]. In our approach, the domain analysis is an important task to define the commonality and the variability of a product line.

Application Engineering

Application engineering refers to the activities of combining the artifacts produced in the domain engineering to build members of a product line. The application engineering focuses on reusing the domain artifacts and exploiting advantages of commonalities and variabilities to develop individual products of a family. During the application requirements phase, various features of a concrete product can be identified that conform to context, e.g., user requirements, execution environment, etc. Based on a set of such features, in the application derivation phase, a set of corresponding available artifacts identified in the domain engineering is selected to build concrete products. The selection can be made through a variability model that presents the commonality and the variability of an SPL. Once the selection decisions are made in the variability model, the variability model is said to be configured. Consequently, the related software artifacts can be deduced to build the final product (or product configuration). This task can be manually or automatically realized using code generation or models composition.

2.2.2.2 Variability Modeling

The basis of SPLE is based on managing the variability and the commonality. Variability management is the most important activity that is performed during the whole product line development cycle [Arboleda 13]. It is considered as a key feature to distinguish SPLE from other software engineering approaches [Bosch 01]. A main issue of variability management in software product line is the explicit representation of the variability [Sinnema 04]. A way to represent efficiently variability is to model it. A review of variability modeling approaches

is in [Chen 09, Czarnecki 12].

FODA [Kang 90] is one of the first approaches to model variability in SPL. This approach unifies the notion of *feature* to represent the commonality and variability in a tree structure, called *feature model*. A feature is abstracted as a capability provided by the system [Elkhodary 10]. It is considered as a unit of software functionality or non-functionality that meets a requirement, implements a design decision, or provides a potential configuration option [Apel 09]. Figure 2.3 shows an example of a feature model. This model contains a root feature that is decomposed into mandatory, optional features, feature groups as or-groups or xor-groups.

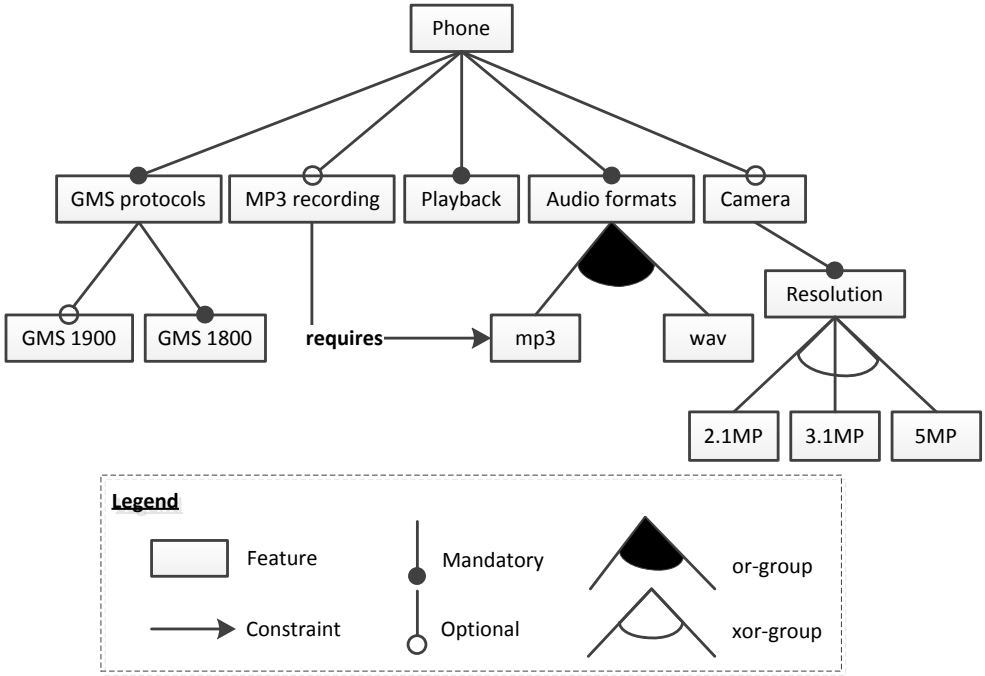


Figure 2.3 – Feature model (adopted from [Czarnecki 12])

The type of features is extended in the cardinality-based feature model proposed in [Czarnecki 05c] with the notion of cardinality of a feature (the clone number), e.g., solitary features with cardinality [1..1] are referred to as mandatory, whereas solitary features with cardinality [0..1] are called optional. Additionally, a solitary feature with cardinality [n..m] indicates that the clone number of the feature can be configured, e.g., a printer may have some clones of cartridge. On the other hand, the cardinality can be applied to feature groups to identify the features number in the group to be selected.

In addition to hierarchical relation among features, cross-tree dependencies among features can be represented. This dependency is defined using two constraints, "requires" and "excludes", i.e., a feature requires/excludes another feature, respectively. For instance, in order to record a mp3 file with the MP3 Recording feature, an application requires the MP3 feature in the Audio Formats feature group.

Moreover, one important characteristic in variability modeling is binding time. It indicates the moment when a feature is selected for the final product during the development process. If features are selected too early, the flexibility of the product line artifacts is less than required. If features are selected later than necessary, the resulting solution is more costly than necessary [Bosch 01].

Feature models are largely used in research and industry. Apart from feature models, a recent approach raised to represent variability is Common Variability Language (CVL). CVL separates clearly between the variability and the software architecture. We will present CVL and its main concepts in Section 2.3.

2.2.2.3 Variability Configuration and Product Derivation

The aim of variability configuration is to allow producing software products under the product line approach. Once variability is identified in the domain engineering, i.e., the variability of the family is defined in architecture and implemented in its source code, various products of the family can be statically deduced by configuring the variability model, or even reconfigured at runtime [Capilla 13].

Product configuration is based on the idea that product derivation activities should be based on the parameterization and/or composition of the SPL artifacts [Perrouin 08]. Most approaches are based on the original feature model proposed by Kang et al [Kang 90]. The mapping between the feature model and the software architecture as well as design artifacts allows to identify elements in the architecture when a selection is made in the feature model. The selection can be realized through various stages of the software development process [Czarnecki 05b]. Finally, a particular product can be derived.

2.3 Common Variability Language

In this section, we present the main concepts of CVL for specifying the variability and the commonality of SPL. CVL is a domain-independent language,

and also an approach for specifying and configuring variability. An overview of the CVL approach is depicted in Figure 2.4. Three models are defined: the base model that allows to model the elements of the architecture, the variability model that models variability in the base model, and the resolution model that is defined to configure the variability model.

Let CVL model denote the three models: the base model, the variability model, and the resolution model. A particular product can be generated by using the CVL execution that takes CVL model as its input. An overview of these three models and the CVL execution is presented in the two following sections.

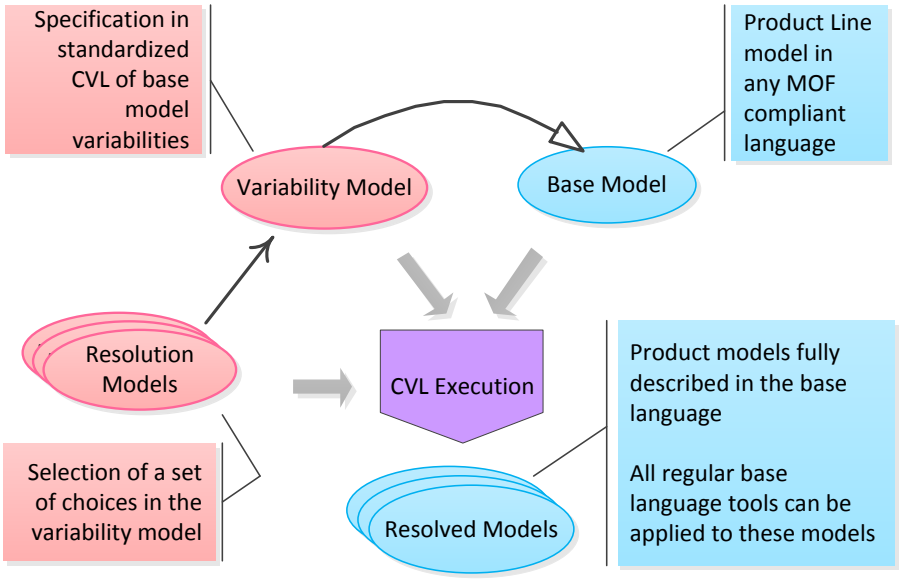


Figure 2.4 – Common Variability Language approach (adopted from [OMG 12])

2.3.1 Specification

Specification is considered as a stage in the domain engineering process. Variability and commonality of an SPL are specified using the variability model, whereas architecture of an SPL is specified using the base model.

2.3.1.1 Base Model

A base model is used to represent concrete elements for building the different products of a family. It can be defined in any MOF-defined language [OMG 12]

or domain-specific modeling language (e.g., UML). It can also be considered as a component-based architecture model, e.g., Fractal ADL [Coupaye 07], ACME [Garlan 00]. As the architecture of a family of products, it embeds variability thanks to alternative elements. A base model is not a consistent nor a runnable architecture. Thus, it needs be configured to identify elements for generating the final product that is runnable.

In our approach, the base model is specified as a component-based architecture. Reconfiguring this architecture consists of adding, removing, replacing components, or modifying parameters in a component.

2.3.1.2 Variability Model

As mentioned in the previous sections, a variability model captures variability and commonality of a product family. It allows making explicit the differences and the similitudes between products in the same family.

In the CVL approach, variability is specified in a variability model conforming to the CVL meta-model. The variability model consists of three parts: the variability specification tree (*VSpec* tree), variation points, and Object Constraint Language [1] (OCL) constraints.

VSpec tree

A variability specification (*VSpec*) is the central concept of variability modeling in CVL. It is an indication of variability in the base model [OMG 12]. *VSpecs* may be organized into a tree, called *VSpec* tree, where the parent-child relationships are defined.

The *VSpecs* can be divided into three kinds: *Choice*, *Variable*, and *VClassifier*. *Choice* requires a binary selection (true/false). *Variable* allows providing a value of a certain type. *VClassifier* allows specifying instance multiplicity of a *VSpec*. It indicates how many instances of it can be created.

In a *VSpec* tree, the parent-child or hierarchic relationships indicate certain constraints on the decision of the nodes such as optional, mandatory, or group multiplicity. The relationships of *Variable* and *VClassifier* to their parent are always mandatory. On the other side, relationship of *Choice* to its parent is either mandatory or optional specified by using a field "isImpliedByParent" in *Choice*. Moreover, a *VSpec* may have a group multiplicity to specify how many children must be resolved. CVL uses the terms of "resolution" to indicate

1. http://www.omg.org/spec/OCL/

a decision for a *VSpec*. If a *VSpec* is resolved positively, the *VSpec* will be configured in the final product.

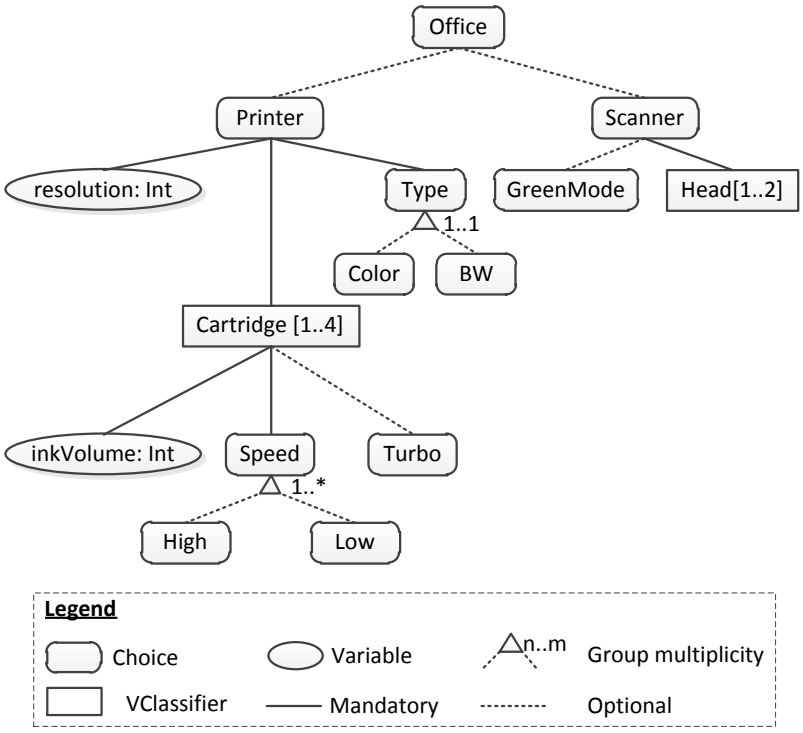


Figure 2.5 – *VSpec* tree (adopted from [OMG 12])

For example, a *VSpec* tree is shown in Figure 2.5. The rounded rectangle, rectangle, and ellipse are used to present *Choice*, *VClassifier*, and *Variable*, respectively. The solid and dotted lines present the mandatory and optional relationships between two *VSpecs*, respectively. If we do not select **Printer** for configuration, i.e., **Printer** is negatively resolved, we must decide negatively for **Type**, there is no instance of **Cartridge**, and no value is assigned to **resolution**, and so on recursively down the tree. On the contrary, if **Printer** is positively resolved, its children must be decided, i.e., **Type** must also be positively resolved, a value should be assigned to **resolution**, and there is at least one instance of **Cartridge**.

The remainder of this dissertation focuses on the *Choice VSpecs* because we work on adaptive software architecture, i.e., the software architecture will be modified at runtime by replacing components thanks to selection or deselection of *VSpecs* in the *VSpec* tree. A *Choice* is resolved positively or negatively, i.e.,

it is decided to TRUE or FALSE. Thus, the *Choice VSpec* is appropriate for representing variability of components in adaptive software architectures.

Variation Points

Variation points link *VSpecs* to elements of the base model affected by the variability. They allow to identify what elements of the base model are removed, added, or/and modified when a *Choice* in the *VSpec* tree is resolved positively. We consider three ways that variation points define the changes of the base model:

- *Existence.* This is a type of variation point in charge of representing whether an object of a link exists or not in the product model. The object and link of a base model can be linked to *ObjectExistence* and *LinkExistence* variation points, respectively.
- *Substitution.* This type indicates that a single object of an entire model fragment may be substituted by another one. The single object and the entire fragment can be linked to *ObjectSubstitution* and *FragmentSubstitution* variation points, respectively.
- *Value assignment.* It is used to represent that a given value is assigned to a given slot in the base model, e.g., *SlotAssignment* variation point.

OCL Constraints

CVL supports the definition of OCL constraints among elements of a *VSpec* tree that cannot be directly captured by hierarchical relations in the *VSpec* tree. CVL presents a basic constraint language - a restricted subset of OCL. Additionally, CVL allows to use other constraint languages, including more complete OCL. Some constraints are usually used such as *implies* and *excludes*. For example, a constraint "A implies B" indicates that if A is resolved to TRUE, B must also be resolved to TRUE.

2.3.2 Configuration

Configuration is taken into account in application engineering process to generate product models from the specified models in the domain engineering. In CVL, the product model is generated by configuring the variability model. This model is configured thanks to a resolution model defined by an application engineer.

2.3.2.1 Resolution Model

A resolution model presents decisions to configure the variability model. When a variability model is configured, each *VSpec* is resolved. The resolution is presented as a tree where each element, called *VSpecResolution*, refers to a *VSpec* of the variability model. According to the three types of *VSpec*, there are three types of *VSpecResolution*: *ChoiceResolution* that resolves *Choice VSpecs*, *VariableValueAssignment* that resolves *Variable VSpecs*, and *VInstance* that resolves *VClassifier VSpecs*.

Figure 2.6 shows a resolution model that corresponds to the *VSpec* tree of Figure 2.5. Each node in the tree is a *VSpecResolution* that refers exactly to a *VSpec* in the *VSpec* tree. In Figure 2.6, `Printer=True` is a *ChoiceResolution* that resolves `Printer` positively; `resolution=1600` is a variable value assignment that resolves `resolution`; `c1:Cartridge` as well as `c2:Cartridge` are VInstances that resolve `Cartridge`, and so on. Finally, `Scanner=False` resolves `Scanner` negatively. So, the children of `Scanner` need not be specified in the resolution model. In addition, a *VSpec* that is not resolved by any *VSpecResolution* of the resolution model can use the default decision specified in that own *VSpec*.

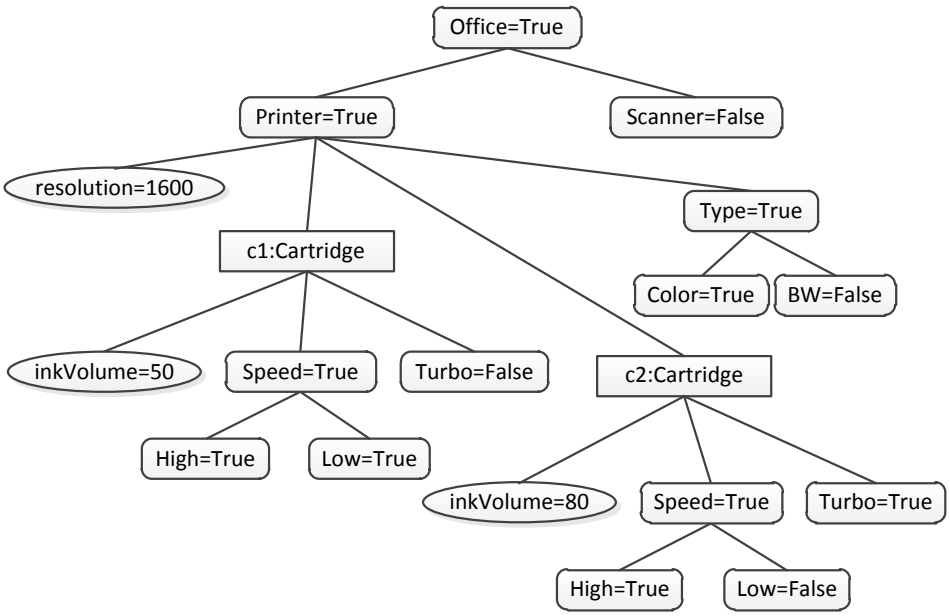


Figure 2.6 – Resolution model (adopted from [OMG 12])

Once all the *VSpecs* are resolved, i.e., the variability model is configured,

the variation points are used to decide which components and attributes in the base model are affected by the configuration to create the product model.

2.3.2.2 CVL Execution

CVL provides a tool, called CVL Execution (Figure 2.4), that takes the CVL model (a variability model, a base model, and a resolution model) as its input. The result of running the CVL Execution is a product model that is specified in the same language of that base model.

Conclusion

CVL offers tools and meta-models to specify variability of a product family but it does not offer a method to specify the variability model and the base model. In addition, the product is generated without runtime variability.

2.4 Software Architecture

2.4.1 Definition

Software architecture is considered as a discipline of software engineering. The description of software architecture allows to effectively represent the complex software system throughout its development, deployment, and evolution/adaptation.

Presently, there is no definition about software architecture accepted widely by the software engineering community. A model of software architecture was early proposed in [Perry 92] as a triple:

$$Software\ architecture = \{Elements,\ Form,\ Rationale\}$$

This means that a software architecture is a set of elements in which each element has a particular form. Elements are distinguished between three types: processing, data and connecting elements. These three types can be consolidated into the two main concepts, *components* and *connectors*. The form presents the way in which the elements are arranged in the architecture. Finally, the rationale presents the motivation for the choice of elements, and the form [Perry 92]. Shaw and Garlan [Shaw 96b] were based on the definition proposed by Perry *et al.* to define software architecture as follows:

"Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns".

In this definition, the composition among architectural elements and the dependencies between them are mentioned. In fact, the elements in architecture are composed in an entity to accomplish the system, i.e., the elements are organized or structured. According to Kruchten *et al.* [Kruchten 06], the following definition was proposed:

"Software architecture involves the structure and organization by which modern system components and subsystems interact to form systems, and the properties of systems that can best be designed and analyzed at the system level".

Recently, Bass *et al.* in [Bass 12] defined:

"Software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both".

Although these definitions focus on various aspects of software architecture, all of them refer to two main aspects, *structural* and *behavioral*. The system structure is described in terms of components, connectors, and configurations. The behaviors are considered as the interactions between system components via connectors to achieve the functional system. They may be described in terms of actions and their relations, behaviors of components and connectors, and how they interact and change, the state of the active system [Szyperski 02]. In this thesis, we focus on the structural aspect including components, connectors, and configurations of the software architecture to cope with architectural adaptation.

2.4.2 Component

A component can be used to represent calculation elements or store units of the system such as clients, servers, databases, user interfaces, etc. [Schmerl 02]. It can be simple/atomic or composite, and its functionality is originated from a simple procedure or a complex application [Moo-Mena 07]. The software component is defined in [Szyperski 02] as:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

The component in this definition is seen as a composition unit with inter-

faces and explicit dependencies. A component may have multiple interfaces in which each defines an interaction point with other component or its environment. Components play a central role for building software systems, or can be integrated by third-parties. In terms of building software systems, Taylor *et al.* [Taylor 09] defined software component as:

"A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context"

According to this definition, a component highlights the role of calculation or/and data of a system. Those calculation and data can be accessed via explicitly defined interfaces. Moreover, a critical aspect of software components makes them usable and reusable according to the required execution context.

2.4.3 Connector

In software architecture, components are mainly responsible for processing or data, or both simultaneously. Another aspect in software architecture is interaction among components. This interaction is represented using connectors. A connector is defined in [Shaw 96a] as:

"Connectors are the locus of relations among components. They mediate interactions but are not–"things" to be hooked up (they are, rather, the hookers-up). Each connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc."

According to this definition, a connector is considered as a mediate element for communication and coordination activities among components, as well as a software element whose properties can be specified. This definition mentions rules for interactions. The interaction aspect of connectors is also mentioned in the definition of Taylor *et al.* [Taylor 09] as:

"A software connector is an architectural element tasked with effecting and regulating interactions among components."

In order to effect and regulate interactions, a connector must provide services. In Chapter 5 in the book of Taylor *et al.* [Taylor 09], a software connector can provide four classes of services as follows:

- **Communication:** provides services for transferring data among components.
- **Coordination:** provides services for transferring control among components.
- **Conversion:** provides services for transforming the interaction required by one component to that provided by another.
- **Facilitation:** provides services that mediate and streamline component interactions such as load balancing, scheduling services, and concurrency control.

On the other hand, connectors can offer extra-functional services, such as logging, transactions, quality of service constraints, and so on. These functionalities are independent of the interaction of component's functionalities and are usually provided by platforms such as CORBA, DCOM or RMI. In fact, connectors are not clearly presented in such platforms. Furthermore, they are not explicitly presented in some component models such as Fractal [Coupaye 07], or iPOJO [Escoffier 07].

2.4.4 Architectural Configuration

In order to accomplish system's objective, components and connectors are composed in a specific way into a set of software elements. Such a set of elements is considered as the system's configuration. Taylor *et al.* defined a configuration as follows:

" An architectural configuration is a set of specific associations between the components and connectors of a software system's architecture."

This means that configuration is a particular structure for a concrete system. It can be represented as a graph in which nodes represent components and arcs represent connectors. A configuration can be built as a complete product. For example, the product model generated by CVL Execution may be considered as an architectural configuration.

2.4.5 Architecture Description Language

Software architectures can be described using an Architecture Description Language (ADL). ADLs aim at specifying a high-level structure of the application rather than the implementation in a specific source code [Vestal 93]. ADLs sup-

port architecture-based development by providing a foundation with standard notions for specifying and describing the software systems [Medvidovic 00].

Many ADLs have been proposed to model and represent software architecture from the 90 years, such as Rapid [Luckham 95], Darwin [Magee 96], Weight [Allen 97], ACME [Garlan 00], and so on. Each of them specifies the architecture system according to its own way. However, all of them are based on the basis concepts of component and connector.

In this thesis, we use ACME to specify the component-based adaptive architecture or the base model in terms of CVL as our ADL. It is used for three reasons: 1) It is a general purpose ADL that supports extensible architectures for different domains, and extensible properties and architectural analyses; 2) It extends the usual component-connector representation with the concept of families, allowing designers to define different architectural variants or styles [Schmerl 02]. 3) It can be extended to support runtime adaptation [Cheng 02b, Cheng 02a].

ACME: an architecture description language [Garlan 00]

An architecture structure defined in ACME uses main types of elements such as components, connectors, systems, ports, roles, and representations.

Each component in ACME can have multiple interfaces called ports, which identify a point of interaction between the component and its environment. Similarly, connectors in ACME also have interfaces which are defined by a set of roles. The roles in the connector allows identifying the role of participants in interactions. For instance, a connector used to represent interaction between two components has two roles such as (caller, callee) or (sender, receiver). Components and connectors can be composed in a set of elements, called system, by attaching component ports to connector roles. It is represented as a graph in which nodes are components and arcs are connectors. In ACME, each component or connector can be represented by one or more details, lower-level descriptions which is called a representation. Connection between components and their more detail is represented by binding.

For example, a client server system can be represented in ACME as Figure 2.7. The system consists of a `Client` and a `server` that can be implemented by `Server 1` or `Server 2`. Each component has a port described as a small black square. The interaction between the client and the server is represented by a connector with two roles, caller and callee. The connector is attached with two ports of `Client` and `Server`. Two bindings are used to map two ports from `Server` to `Server 1` and `Server 2`, respectively.

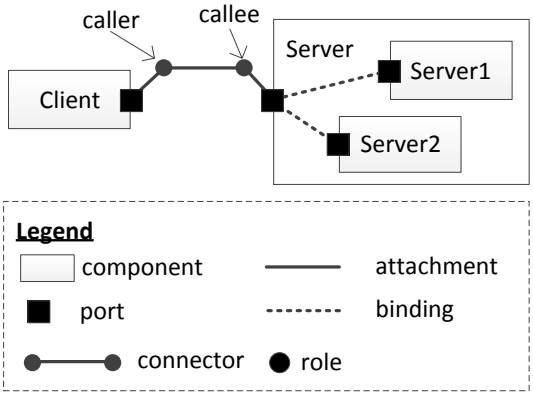


Figure 2.7 – A client-server system represented in ACME

2.5 Architecture-Based Software Adaptation

The previous sections present an overview of methodologies about software development and concepts related to variability and software architecture. The methodologies are considered as one solution to develop adaptive software. They are the basis of Dynamic Software Product Line (DSPL) engineering that emerged as an efficient way to deal with runtime software adaptation [Cetina 08b]. In this section, we describe the relation between DSPL and software adaptation to highlight the role of DSPL in software adaptation development. In addition, the terminologies and principles related to software adaptation and architecture-based adaptation are introduced. Finally, adaptation mechanisms are studied on how to ensure dynamic consistent adaptation.

2.5.1 Terminologies

Software adaptation is the capacity of a software to adapt to a changing operating environment. The operating environment includes anything observable by a software system, such as end-user input, external hardware devices and sensors, or program instrumentation [Oreizy 99]. Indeed, when the operating environment changes, the system needs to be modified to meet the new conditions [Cheng 09]. Through such an adaptation, the software system moves from a current version to a new one.

A characteristic of adaptation is the moment when the changes can be performed. Based on the adaptation time in the development process, two adaptation types can be identified: *static adaptation* and *dynamic adaptation* [Canal 06, Geihs 09]. The former refers to the modifications that are performed

during design, and development time. Artifacts touched by the static adaptation can be the design model, the documentation or the source code. If the static adaptation corresponds with activities related to the maintenance [Chapin 01], the system needs to be stopped, and its services are interrupted. The latter refers to the changes of software system that happens during system execution. This adaptation type maintains the continuity of services assured by the system. Moreover, it must also ensure the consistency of the system during and after adaptation. According to the definition of Oreizy *et al.* [Oreizy 98], the dynamic adaptation is defined as realizing modifications in a system without recompilation during runtime. Compared to the static adaptation, the dynamic adaptation significantly reduces the disruption of services provided by the system.

Software systems that perform such dynamic adaptation are called *adaptive* or *self-adaptive* software systems. They are defined as a class of software which is able to dynamically modify (at runtime) its own internal structure or/and its behavior in response to changes in its operating environment [Oreizy 99, Cheng 09, de Lemos 13]. In fact, it does not exist a clear separation between adaptive and self-adaptive. An adaptive system is called self-adaptive if it observes its execution environment, decides adaptation, and modifies the system by itself, i.e., without the intervention of external agents, e.g., user intervention. The *self-adaptive* terminology is closely related to the self-* properties of software systems such as self-managing, self-healing, self-optimizing, and self-configuring [Salehie 09]. On the other hand, an adaptive system may be intervened by external agents that can decide and/or realize the adaptation on the running system. In this thesis, we focus on the *dynamic* or *runtime* aspect of software adaptation.

Depending on the degree of automation, we distinguish three types of adaptation: *manual adaptation*, *automatic adaptation*, and *semi-automatic adaptation*. The former requires an engineer to decide, plan and implement manually adaptation mechanisms, e.g., [Buisson 15]. The second one refers to the self-adaptive systems, e.g., [Geihs 09, Pascual 14]. Finally, in the third one, some activities such as monitoring execution environment and deciding adaptation are realized by an external agent, while the planning and executing adaptation are realized by the system itself, e.g., [Phung-khac 10].

Adaptation is realized by adaptation controllers. The adaptation is called to be centralized when a centralized controller is used. Otherwise, it is distributed. In the distributed adaptation, several entities coordinate with each other to manage and control the adaptation process. In this thesis, we focus on adaptive software systems whose adaptation is controlled at runtime by a centralized controller integrated in it. This controller will receive a decision from an external

agent for adaptation.

An adaptation can change the functionality of a system or not. This refers to the terms of *functional/non-functional adaptation*. A functional adaptation will change the software system functionality, i.e., services provided to the users can be modified, added, or removed from/to the software system after the adaptations. According to Parlavantzas *et al.* [Parlavantzas 05] and Bencomo *et al.* [Bencomo 08], functional adaptation refers to the notion of extensibility as the ability to add new functionality. On the other hand, non-functional adaptation leads to changing the characteristics of the system such as quality of service, performance, security, etc. For example, the adaptation proposed by Pascual *et al.* [Pascual 14] is considered as non-functional adaptation for optimizing its activities. In this thesis, we consider non-functional adaptation that replaces an architectural variant by another one.

2.5.2 DSPL and Software Adaptation

The term DSPL was firstly introduced in [Kim 04] as an approach where new products can be produced at runtime without stopping the running system. According to [Hinchey 12], DSPLs extend traditional SPLs approaches by moving their change capabilities to runtime, i.e., a new product version can be derived at runtime by modifying or reconfiguring the running system. Generally, the aim of DSPL focuses on developing software products that can be adapted at runtime to cope with the more frequent changes of the operating environment such as user requirements, end-user input, external hardware devices and sensors, or program instrumentation.

DSPLs have been proposed to use the techniques in SPL engineering to build adaptive software systems [Hallsteinsen 06, Bashari 13]. They are considered as an efficient way to deal with runtime product adaptation. In comparison with traditional SPL, DSPL highlights the dynamic aspect, i.e., software artifacts are used to dynamically replace parts of the running product based on variability specified in SPL. Thus, an adaptation mechanism should be provided to cope with runtime variability in the product. In DSPLs, observing the execution context, deciding and controlling the adaptation are critical tasks. The application engineer, the user, or the application itself can carry out manually or automatically these tasks [Hallsteinsen 08].

The software artifacts in DSPLs that are not selected for configuring the initial product should be integrated into the product. They can potentially be used at runtime to cope with new requirements. Thus, such artifacts must be available at runtime. By default, all artifacts in DSPL are available in the

product for adaptation. We have not found any approach that identifies artifacts that are not required for a particular product.

2.5.3 Adaptation Control Loop Model

In a classic system, i.e., non automatic, a system administrator is responsible for observing the system, analyzing new requirements, and deciding the actions to maintain the services provided by the system. Once the system is designed and deployed, it can not be repaired without stopping it. Such a system is designed as an *open-loop* system. On the other hand, an adaptive software system is provided an external adaptation mechanism based on *closed-loop control* to monitor and control adaptation at runtime [Garlan 04]. Figure 2.8 shows a closed-loop control that includes adaptation mechanisms to monitor and modify dynamically the system. Such kind of control can be considered as a feedback control loop.

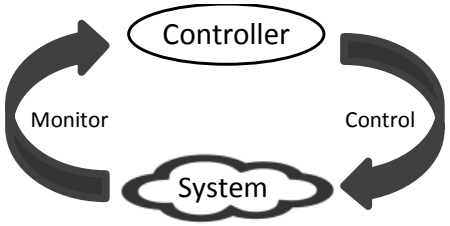


Figure 2.8 – A closed-loop control model (adopted from [Garlan 04])

A feedback is information that is translated back into an action performed onto the system from which it has originated [Křikava 13]. A feedback control loop includes four key activities: collect, analyze, decide, and act, as depicted in Figure 2.9 [Dobson 06]. The loop starts with the *collection* of relevant data reflecting the system and its environment from sources such as environmental sensors and user context. Next, the system will *analyze* the collected data. Based on the results of analysis, a *decision* is made about how to adapt to reach a desirable state. Finally, the system must *act* to realize the adaptation decision via available actuators and effector [Cheng 09].

The generic model of a feedback loop is sometime also referred to as *autonomic control loop* [Brun 09] or *adaptation loop* [Salehie 09]. The idea of this model originates from the autonomic computing research community to enable self-management of system [Kephart 03]. It is known as Monitor-Analyze-Plan-Execute loop (MAPE or MAPE-K with a shared knowledge component) [Kephart 03]. Figure 2.10 shows the MAPE loop that distinguishes between

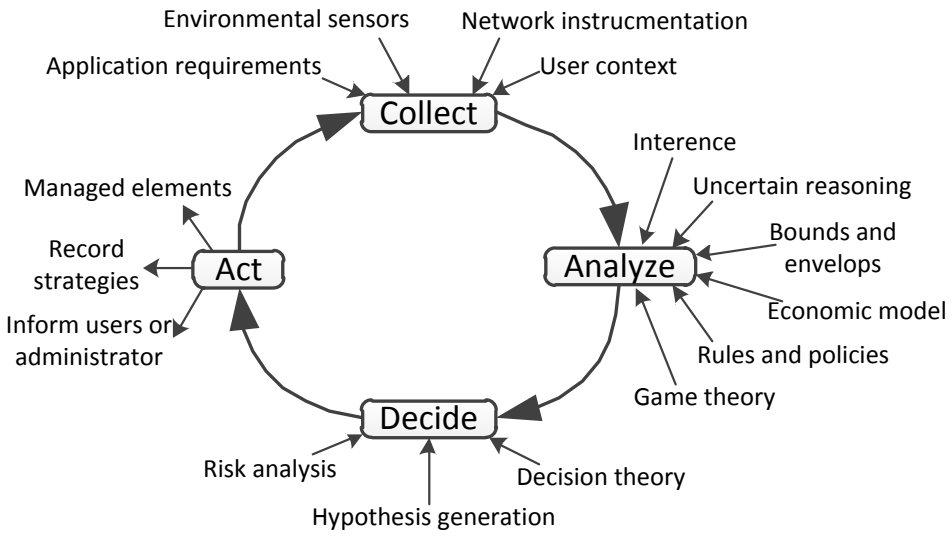


Figure 2.9 – Feedback control loop model (adopted from [Dobson 06])

the managed element (the running system) and the autonomic manager. The managed element may be either entire system or a component within a larger system. The autonomic manager includes four elements that observe system parameters, analyze them, plan actions and execute them. The knowledge element shared between these elements is standard data such as models, databases, or dictionaries. It can also be shared among autonomic managers [IBM 05].

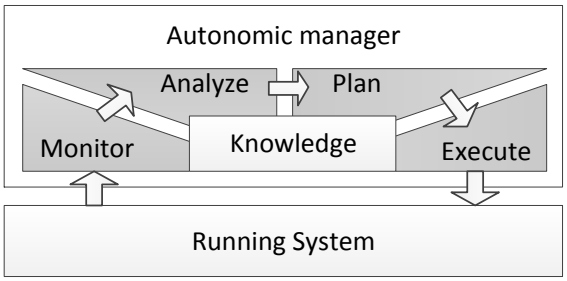


Figure 2.10 – MAPE-K loop (adopted from [Kephart 03])

- Monitor: The monitoring phase captures properties related to the managed elements and its external environment. Based on the knowledge, the monitor identifies what it observes and collects. The data can be collected through sensors and transforms the current context to a form that can be

manipulated by other phases.

- Analyze: Based on the context information provided by the monitor phase and the knowledge, the analyzing phase evaluates changes and determines a new target system and offers change decisions.
- Plan: The planning phase receives the change decisions and make an adaptation plan based the knowledge and the new target system. The plan provides necessary actions for adaptation.
- Execute: The plan is provided to the executor to realize adaptation actions. The executor can observe the ongoing activities in the current system to decide an appropriate moment to apply the execution of adaptation actions.

In this thesis, we focus on the activities of planning and executing adaptation actions. We adopt the MAPE-K loop in which the executing element is called *Reconfigurator*. This *Reconfigurator* can access the knowledge repository that contains data such as the variability model, the deployment model, etc, and receives a reconfiguration plan generated from the planning element to execute adaptation on the running system.

2.5.4 Architecture-Based Adaptation

Software architectures are considered as design-time artifacts. Architectural models are specified by using ADLs with explicit components and interactions among them, and analyzed by using design-time tools. The idea of architecture-based adaptation is to maintain these models and tools at runtime to be used as a basis for adaptation [Cheng 02a].

Architecture-based software adaptation provides adaptation mechanisms based on architectural models to monitor and adapt the running system. This kind of adaptation uses a closed-loop control to manage and control adaptation as mentioned in [Oreizy 99, Cheng 02c, Garlan 04].

In [Heimbigner 02], configurable runtime architecture models are used to describe the structure of the software system. They are moved from design to runtime to define multiple configurations for the architecture. Such models are attached to the software system to provide the basis for defining and monitoring the context. Moreover, they are used for reconfiguring the system as well.

Architecture-based adaptation facilitates managing and realizing adaptations using architectural models. The architectural models allow to represent explicitly components and their constraints. For this reason, one of the main

advantages of using architectural models is that they support to validate new configurations of the system as well as ensure the system consistency and integrity [Oreizy 98, Morin 09b].

2.5.5 Consistent Dynamic Adaptation

A critical requirement of adaptation is to ensure the system consistency, i.e., the correct completion of ongoing activities must be preserved, the adaptation process does not break component dependencies, and the state of current components must be transfered to the new ones [Li 12]. An adaptation that ensures such requirements is called consistent dynamic adaptation.

Two of the three most important issues of consistent dynamic adaptation indicated by Ghafari *et al.* [Ghafari 12b] are reaching a "safe status" [2], and transferring the internal state of entities which have to be replaced. The former indicates the moment when components can be replaced. The latter focuses on the state transfer between the current and the new configuration.

In this section, we present concepts and an overview of existing approaches related to these issues in order to understand how to manage them.

2.5.5.1 Definition of Transaction

Consistent dynamic adaptation is closely related to the notion of *transaction*, since both are looking for the consistent status of a system.

According to Kramer and Magee in [Kramer 90], a transaction is defined as *"an exchange of information of two and only two [components]*[3]*, initiated by one of the [components]"*. It consists of a sequence of messages exchanged between two components that completes in bounded time. Similarly, Rasche *et al.* [Rasche 03] defined *"a transaction is a sequence of one or more message exchanges over a connection"*. These definitions highlight a cycle of communication messages without the computation activities of the components and consider a transaction via two connected components. A transaction is initiated when a component invokes a service on other components.

Figure 2.11 shows three components, `Initiator`, `Dispatcher`, and `Receiver`, participating in two transactions `T1` and `T2`. According to the transaction def-

2. We use "status" as mentioned in [Vandewoude 07] instead of "state" to differentiate from internal state, e.g., value of variables, properties of components, or data, etc. Status indicates an appropriate moment for adaptation, whereas, state indicates state space.

3. In this thesis, we use the term of *[components]* instead of *nodes* that some authors used.

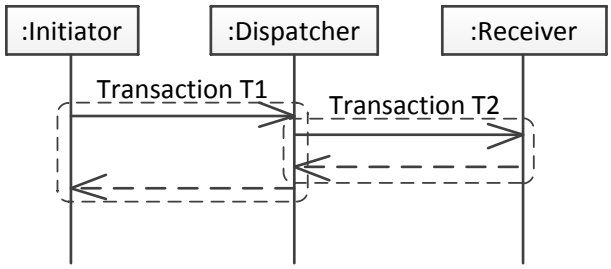


Figure 2.11 – Example of transactions according to the definition in [Kramer 90]

inition of Kramer and Magee, `T1` relates to two components (`Initiator` and `Dispatcher`) while `T2` relates to the `Dispatcher` and `Recipient` components.

On the other hand, Ma *et al.* in [Ma 11] define a transaction as *"a sequence of actions executed by a component that completes in bounded time"*. Actions include local computations on a component and its message exchanges. An example is represented in Figure 2.12. It includes three transactions instead of two as in Figure 2.11. Each transaction is engaged in a component.

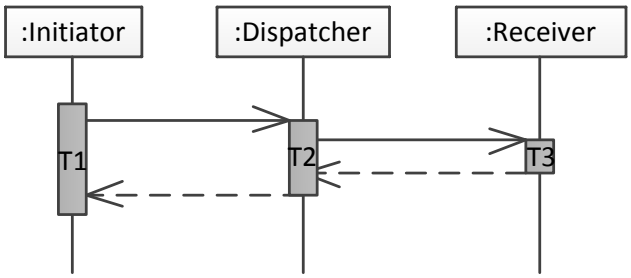


Figure 2.12 – Example of transactions according to the definition in [Ma 11]

A transaction can be initiated by another transaction. Such a transaction is called *consequent transaction*. Kramer and Magee indicated that the transactions in a system can depend on each other. A *dependent transaction* is a transaction whose completion may depend on the completion of other *consequent transactions*. On the other hand, a transaction is *independent* if its completion does not depend on any other transactions.

According to the definition of dependent transactions, Figure 2.11, `T2` is called consequent of `T1`, i.e., `T1` completes iff T2 has finished. Finally, `T2` is an

independent transaction. On the other hand, in Figure 2.12, `T1` depends on `T2`, and `T2` depends on `T3`. `T3` is the consequent transaction of `T2`. Similarly, `T2` and `T3` are the consequent transactions of `T1`.

We adopt the transaction definition in [Ma 11] and extend it towards many components engaged in a transaction. This extension will be presented in Chapter 5. However, to explain the existing approaches in this section, we adopt their definitions.

2.5.5.2 System Consistency

The adaptation process must preserve the system consistency. A system is consistent if the correctness of ongoing activities of the system is not affected by adaptation, and the state of the whole system is guaranteed. The system consistency is distinguished into local consistency and global consistency [De Palma 01, Ketfi 02]. The former is related to one component independently of the others, whereas the latter concerns the whole system.

Local Consistency

Local consistency is considered as the local state of a single component. It is related to the inner computation of a component [De Palma 01] and interactions between the component and other components [Chen 02]. The local state is consistent if it is maintained during reconfiguration, i.e., the inner computation and its interactions are preserved.

According to [De Palma 01], in order to maintain local consistency, the reconfiguration must take into account the following issues: the reference preservation, the state one, and the state of communication channels. Indeed, when replacing a component by another one, all references of other components that require services of the replaced component must be guaranteed. Secondly, the local state of component must be preserved. Finally, the last issue is related to communication channels. When realizing adaptation, some communication messages may still be in transit. Thus, such messages should be taken into account for adaptation. If one of these issues is not satisfied, the component becomes inconsistent.

Global Consistency

In a system, the local consistency is not enough to ensure the system consistency: global computations must also be preserved consistent in spite of adaptation [De Palma 01]. The global computations concern globally distributed

transactions, where each transaction is performed by a distributed component that is liable to be reconfigured [Li 11].

A system guarantees global consistency before and after adaptation if the adaptation ensures the local consistency of all components and the global computations. In our approach, the global consistency is addressed by managing transactions to identify a safe status, and transferring state in the system.

2.5.5.3 Safe Status

Safe status is an appropriate moment and also a condition which allows an adaptation process to ensure global consistency. Indeed, an existing system has potentially ongoing activities that should not be canceled or aborted by adaptation. On the other hand, changes by adaptation should not lead to an inconsistency in the system. Therefore, the replaced components should be put in the safe status that ensures no inconsistencies arisen in the system.

Detecting the moment when the safe status is reached is a difficult task. This task is considered as the key to ensure consistent dynamic adaptation. Kramer and Magee [Kramer 90] were the first to consider this problem and abstract the status of a system into a set of different status for each component. They consider two main status for each component, active and passive, whose definitions are given as follows:

Definition 1. ***(Active status)*** *A component in the active status can initiate, accept, and service transactions.*

Definition 2. ***(Passive status)*** *A component in the passive status must continue to accept and service transactions, but*

1. *it is not currently engaged in a transaction that it initiated, and*
2. *it will not initiate new transactions.*

Kramer and Magee identify that a passive status is necessary but insufficient for updatability as a component may still be processing transactions that were initiated by other components. Thus, they propose a stronger concept:

Definition 3. ***(Quiescence)*** *A component is quiescent if:*

1. *It is not currently engaged in a transaction that it initiated,*
2. *It will not initiate new transactions,*
3. *It is not currently engaged in servicing a transaction, and*
4. *No transactions have been or will be initiated by other components that require service from this component.*

In the quiescent status, the status of a component is both *consistent* and *frozen*. It is consistent in that the component does not participate in partially completed transactions, and is frozen in that the component state will not change as a result of new transactions [Kramer 90].

In order to achieve the quiescent status of a component `C`, we must ensure that no transactions have been or will be initiated by other components that require services of `C`. This means that the following conditions must be ensured:

- Component `C` must be passive.
- All initiated transactions and their consequent transactions that have been realized or will be realized by the component `C` must complete.
- All components that can initiate transactions that result in consequent transaction on `C` must be passive.

The quiescent criterion is a sufficient condition for a component to be safely modified in dynamic reconfiguration. However, to reach and maintain the quiescent status, **all** components that can directly or indirectly initiate a new transaction on this component must be in passive status. In the worst case, all the components in the system are passive, which may lead the system to an unavailability status. Consequently, quiescence often causes significant disruption to the running system that may not be acceptable. This is a serious drawback with respect to the impact of changes in the system [Arnold 96].

Vandewoude *et al.* proposed the concept of tranquility, as an alternative to quiescence [Vandewoude 07], and defined it as follows:

Definition 4. ***(Tranquillity)*** *A component is tranquil if:*

1. *It is not currently engaged in a transaction that it initiated;*
2. *It will not initiate new transactions;*
3. *It is not actively processing a request;*
4. *None of its* ***adjacent*** *components are engaged in a transaction in which it has both already participated and might still participate in the future.*

Similar to the quiescent status, the tranquil status is based on the passive status (conditions 1 and 2) and the condition 3. However, the tranquil component only considers its ***adjacent*** components, i.e., the tranquility reduces the number of passive components. The component can be safely replaced if it has not yet begun a transaction, or consequent transactions.

In order to maintain the tranquility of a component as soon as reaching tranquil status, all interactions between that component and its environment must be blocked. This is not taken into account in the quiescence because all components that may directly or indirectly initiate new transactions on the component are passive and remain passive until they are explicitly reactivated.

Although the tranquility status is a sufficient condition for preserving system consistency during reconfiguration, this criterion does not work safely. It only ensures local consistency without the global consistency. Thus, the work in [Ghafari 12a] is based on managing a sequence of transactions to find a safe status for a global consistency.

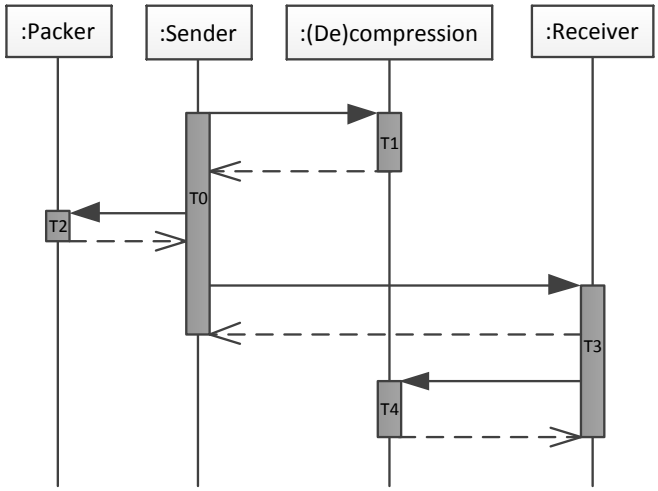


Figure 2.13 – An example for the status management (adopted from [Ghafari 12a])

Figure 2.13 shows a sequence diagram to illustrate a sequence of transactions. In the figure, the `(De)compression` component consists of two services of compressing and decompressing messages. Suppose that the adaptation scenario is replacing the `(De)compression` component by another one. In order to ensure the global consistency, all messages that have been compressed by the `(De)compression` component (transaction T1) must be decompressed by the `(De)compression` component (transaction T4) before replacing it. That indicates that there is a transactional dependency between T1 and T4. Although the approach in [Ghafari 12a] is interested in the transactional dependency, their approach is an ad-hoc one and does not propose a systematic approach to manage such dependencies. More detail of this approach in [Ghafari 12a] are represented in Section 3.6.

If the `(De)compression` component is divided into two components, `Compression` and `Decompression`, applying the quiescent and tranquil criteria does not ensure the global consistency when replacing these components, because the quiescent and tranquil criteria are applied on single components without the transactional dependency. This will be explained in Section 5.2.1.1.

2.5.5.4 State Transfer

The state of a system includes the local state of all components and all messages in transit [Ma 11]. The local state consists of all information such as component properties, data of a component in the system. According to Grondin *et al.* [Grondin 08], a system state is defined as the set encompassing values of all variable attributes of all roles in all configurations.

State transfer is an important issue of dynamic adaptation [Ghafari 12b]. It is considered as a process of capturing the runtime state of a component or a group of components and using this state to initialize a new version.

In the literature, two existing techniques related to state transfer are mentioned in [Vandewoude 05], direct state transfer, and indirect state transfer. For the former, the state of the current version is directly used by the replacing version that has the capacity to interpret and convert state from the current version. For the latter, the current version exports its state in an abstract representation that is used by the replacing version.

The most difficult task of state transfer is to identify the correspondence among states of the component versions. As the semantic of this state is component-specific, the state migration task can never be completely automated [Vandewoude 03]. Hence, a human interaction will always be required. In this thesis, we use a state transfer model to represent the state mapping among components.

2.6 Summary

In this chapter, we have briefly introduced the background and concepts recommended for understanding this dissertation. We have given an overview of methodologies of software development based on models. Particularly, SPL arises as an efficient engineering for developing software in which variability modeling and software architecture specification play a critical role. For the variability modeling, there are different approaches such as feature model, OVM, or CVL, in which CVL clearly separates variation points from the variability

specification, and the software architecture. The software architecture has also been presented in this chapter. It can be represented by using ADLs. In this chapter, we have briefly presented ACME - an architecture description language that is used in this thesis.

In this thesis, we are interested in the architecture-based adaptation. Thus, we have provided the concepts related to the architecture-based software adaptation, as well as developing adaptive software architectures. Particularly, DSPL is considered as an efficient way to develop adaptive software architectures. On the other hand, guaranteeing the system consistency during adaptation plays an important role in adaptation.

This thesis is based on DSPL to develop adaptive software architectures in which CVL is used to specify the variability and commonality via the variability model. In CVL, a base model is used to specify the software architecture. The variability and the base models are used throughout our development process. At runtime, these models are used as a support for planning the dynamic adaptation. Additionally, other issues such as state transfer, and consistent adaptation should be also taken into account when realizing adaptation.

In the next chapter, we will make a survey of various works that are related to ours. We use the challenges identified in Section 1.2 to analyze and evaluate them. This allows us to make a comparison to find advantages and limitations and to position our contributions.

Chapter 3

Related Software Adaptation Approaches

Contents

3.1 Chapter Overview

In this chapter, we discuss approaches related to adaptive software development process. A description of the existing approaches is performed based on the challenges defined in Section 1.2 consisting of: 1) Modeling variability and commonality for adaptation; 2) Configuring and automatically building adaptive architecture; 3) Supporting state transfer; 4) Automatically planning adaptation; 5) Ensuring consistent dynamic adaptation. This description is shown from Sections 3.2 to 3.6. Finally, Section 3.7 provides an overview of a comparison of these existing approaches and their limitations.

3.2 Modeling Variability and Commonality

Modeling variability and commonality in SPL is considered as an effective way to build systems that can be changed to meet various conditions and requirements. A technique to build adaptive software systems based on specifying variability has emerged as mentioned in the review of Capilla *et al.* [Capilla 14]. They presented many various ways to describe the variability in which the variability modeling based on models is a popular method. There are three main proposals used for modeling variability such as the feature model [Kang 90], the orthogonal variability model [Pohl 05], and recently the common variability language [Haugen 13]. In this section, we use these proposals to present the variability and commonality modeling.

Based on Feature Models [Kang 90]

Most exiting approaches are based on the feature model proposed in [Kang 90] to specify variability for adaptation. In [Trinidad 07], a feature model is used to represent variability for DSPLs. By assuming that a feature corresponds to a component in the architecture, this approach proposes a direct mapping from the feature model to the component-based architecture model. The component-based architecture model includes the core architecture and the "dynamic" one. The core architecture maps to mandatory features while the dynamic architecture to optional features. Thanks to the mapping between features and components, when configuring the feature model, a product can be determined.

Authors in [Cetina 08a, Cetina 13] also used feature models and proposed extensions to represent variability of autonomic systems, called FAMA feature model [Benavides 05]. They use a DSL to specify the product architecture, the PervML language [Muñoz 05]. In [Cetina 13], the mapping between the feature model and the product architecture is defined by the "superimposition operator", whereas [Cetina 08a] uses a realization model to map features in the FAMA feature model to PervML elements.

In FamiWare [Gámez 11], feature models are used to represent variability and commonality of various product configurations. A *feature mapping* is defined to represent the correspondence between features and components. A product architecture can be determined by using a set of parameters that represents the context. Corresponding to various context conditions, different configurations can be identified thanks to variability specified in the feature models.

Gomaa *et al.* [Gomaa 07] introduced a modeling approach to design evolutionary and dynamically reconfigurable software architectures. Feature models

are built and analyzed to identify variability and commonality before the implementation of the system. After the implementation, the models and the system co-exist and evolve. The feature models are dynamically analyzed at runtime to help determine the dynamic impact of each feature on the software architecture. Variation points in the feature model can be realized thanks to "plug-compatible" components, and component interface inheritance in component-based software architectures.

In FUSION [Elkhodary 09, Elkhodary 10], authors use a feature model to abstract capabilities provided by a system. According to this approach, a feature represents an abstraction of elements in the architecture, i.e., it maps to a subset of the software system architecture. Based on feature groups (optional features that are defined in the feature model) variability can be identified.

Unlike the above approaches, Lee *et al.* [Lee 06] proposed a feature-oriented approach to develop dynamically reconfigurable products by analyzing features. This approach is based on feature model to present variability and analyze feature binding units. A feature binding unit consists of a set of service features in which each feature represents a major functionality of the system and may be added or removed as a unit. This approach provides a guideline to design architectural components from the feature binding units. Based on the variability in the feature model, adaptation can be achieved at runtime.

Similarly, Phung *et al.* [Phung-khac 10] proposed the adaptive medium approach, called adaptive medium approach, to develop adaptive distributed applications. This approach separates business logic from the adaptation aspects of the applications. In the business logic, this approach extends the feature model to represent variability and commonality. The model is then refined for generating different products as members of an SPL. Finally, the generated products are composed together with an adaptation medium for performing adaptation.

On the other hand, some approaches are based on context-aware model to enrich the feature model. Authors in [Saller 13, Saller 15] proposed an association between feature models and context aware models to design DSPLs. The feature models are enriched with context information specified in a context model to reason about potential context changes. Based on the feature model and the context aware model, a transition system is derived that defines context-aware reconfiguration behavior. Each state in the transition system is a valid configuration. It provides necessary information to perform adaptation at runtime.

Similarly, in [Mizouni 14], a framework is proposed to build context aware and adaptive mobile applications based on feature modeling and SPL concepts.

The feature model is used to represent variability and commonality of SPL in which each feature associates with an execution context. This approach distinguishes the features into three feature groups, critical, important, and useful. The important and useful features are considered as variation points. Each feature refers to a component or a composite structure in the architectural model. At the end of the design time, features in the feature models are selected to derive members of SPL. Each member is associated with a context of use. In terms of implementation, components are implemented as a set of services of application based on the OSGi component model. A context decision module integrated in the framework decides an appropriate member according to the corresponding context.

Recently, authors in [Mauro 16] proposed the HyVar approach for building adaptive systems. They use feature models enriched by contextual information and propose a reconfiguration engine, called HyVarRec, that can compute the new configurations. The features are incorporated with concrete code artifacts that have to be assembled to produce the actual software products.

Nieke *et al.* in [Nieke 17] proposed an integrated tool, called DarwinSPL, that allows representing three dimensions of variability, spatial variability, contextual variability, and temporal one. Product configuration is considered as spatial variability. The spatial variability includes configurable functionality of a software system in terms of different features defining a set of all possible configurations. The spatial variability is captured in feature models. Each feature in the feature model is referred to its corresponding implementation artifact.

Based on Orthogonal Variability Model (OVM) [Pohl 05]

Besides feature models defined by Kang *et al.* [Kang 90], Pohl *et al.* [Pohl 05] proposed Orthogonal Variability Model (OVM) to represent variability. OVM clearly defines variation points and variants in a variability model that correspond with feature group and feature in feature model, respectively. Moreover, OVM allows to define the mapping to development artifacts via artifact dependencies specified in the variants and the variation points.

Particularly, Bencomo *et al.* in [Bencomo 08] also use OVM to specify variability. This approach proposed a domain specific language (DSL) to specify architecture model, OpenCOM DSL [Bencomo 06]. A particular product architecture consists of selecting a set of elements in the architecture model. Each variant in the variability model refers to a particular product architecture.

Other techniques for adaptation are based on aspect-oriented modeling (AOM). In DiVA [Morin 09b], authors proposed an approach to leverage AOM and

MDE to manage variability at runtime. A variability model is used to represent possible variants of the system. Each variant can be considered as an aspect model. Depending on a given context, a set of aspect models can be selected and dynamically woven into an architectural model of the system at runtime. This solution of managing variability using AOM was also used in [Morin 08, Morin 09a] where the aspects are woven into the architecture model presenting mandatory components to build various product architectures.

Parra *et al.* [Parra 11a, Parra 11b] (CAPucine) defined a feature model conforming to OVM meta-model. They combine the feature model with aspect models. The aspect models describe elements, cores and/or aspects, of any product. Each core is specified using a metamodel that consists of the basic elements of a component and service-based application. The aspects are designed for modifying the elements of the core by adding and deleting new elements at specific points (point cut) in the core structure. When configuring the feature model, core and aspect elements in the aspect models can be identified. These elements are woven with each other to build a particular software product.

Based on Common Variability Language [OMG 12]

A recent approach is proposed in [Haugen 08, OMG 12] as a new way to represent variability separated from software architecture, called Common Variability Language (CVL). Authors in [Pascual 14, Pascual 13] presented an approach that uses CVL to develop adaptive software systems. The system is described by variability models and base models. The mapping between the variability and base models is done using variation points. Similarly, Cetina *et al.* in [Cetina 09b] used CVL to specify variability. This approach uses the PervML DSL [Muñoz 05] to specify the architecture model like [Cetina 08a, Cetina 13].

Work in [Gámez 15] is based on CVL to represent variability as well. This approach uses UML to present the base model. Constraints between the context and variants are defined in OCL in the variability model. When the context changes, a new resolution model can be identified for producing a new product. Similar to [Gámez 11], this approach is also based on the FamiWare middleware to build adaptive software.

Other approaches

In MADAM [Floch 06, Geihs 09], authors proposed an approach to develop adaptive software based on a particular component framework. The framework describes a composition of component types. Each type is considered as a variation point where various component implementations can be plugged in

to achieve variability. Thanks to that, the variability can be identified and used to realize adaptations. A component architecture model is considered as a variability model in MADAM.

Discussion

Modeling variability and commonality for adaptation plays an important role in an adaptive software development process. It allows to exactly describe variation points where changes can be realized in software architectures. However, the feature model proposed by Kang *et al.* [Kang 90] unifies the notion of features in which the concepts of variants and variation points are not clearly distinguished. This drawback is better handled in OVM that defines a feature model with the concepts of variant and variation point. Comparing to the feature model proposed by Kang et *et al.*, OVM explicitly distinguishes between the concepts of variant and variation point. However, in OVM, both variant and variation point are considered as nodes in the model. CVL provides a better way to model variability and separate variation points from variability. CVL was used to develop adaptive software in some approaches such as [Cetina 09b, Pascual 14, Gámez 15]. Unfortunately, no existing approaches provide a guideline to help engineers on how specify the variability model and software architectures for building adaptive software architectures.

3.3 Configuring and Automatically Building Adaptive Architecture

This section briefly introduces related approaches in which variability model is configured and a particular product can be automatically generated from a set of information specified and artifacts implemented at design time. In order to configure variability, a set of information about feature selections must be determined. It can be defined and generated from collecting context information, or user requirements.

In particular, the approach in [Pascual 14] uses CVL to specify the variability. A resolution model is used to configure the variability. It is represented as a vector that is generated by using the DAGAME optimization algorithm to fit the execution context. Its elements contain '1' or '0' values corresponding to selected or unselected *VSpecs* in the *VSpec* tree, respectively. When the context changes, a new vector can be generated at runtime. For a given vector, a new product is generated thanks to a dynamic reconfiguration service, and adapted to the current system.

FUSION [Elkhodary 09, Elkhodary 10] is also based on vectors to define a product configuration. Feature selections are specified in the vectors by an engineer to build adaptive software systems. Metrics collected from the running system are analyzed by the FUSION framework that decides a new optimized configuration at runtime. This configuration allows to determine a new product that will be adapted to the running system by the framework.

In FamiWare [Gámez 11], based on a set of parameters of context and constraints between them, and features in feature model, the feature model can be dynamically configured. Then, a product can be automatically generated by applying model-driven and SPL engineering techniques. At runtime, monitoring services capture the possible elements of the context that may change. So a new configuration can be identified and a new product deduced. A defined plan allows to adapt the current product to the new one. Similar to [Gámez 11], the approach in [Gámez 15] is also based on the context to define a resolution model for configuring the variability model. A new resolution model can be generated at runtime when the context changes.

In MADAM [Floch 06, Geihs 09], an architecture model is used to represent system architecture whose components are annotated with properties. These properties qualify the services offered or needed by components, e.g., they can represent system context such as the memory or the network characteristic. Based on the context description, a product configuration can be identified conforming to a concrete context by the MADAM middleware.

Based on the influences of the context on configuration options, in [Nieke 17], different features can be selected for a configuration by an engineer using a tool provided in the approach. A product can be built by composing implementation artifacts implemented in domain engineering corresponding to the configuration. In DarwinSPL, a reconfigurator, HyVarRec in [Mauro 16], is used to calculate a new configuration thanks to context information provided by users. The new product corresponding to this configuration is adapted to the current product by HyVarRec.

Approach in [Saller 13] used a context model mapped to a feature model. According to requirements of the context, a set of features can be selected to build a product. Due to changes in the context, a constraint solver is used to generate new configuration at runtime. The constraint solver takes the feature and context models as its inputs, its output is various products satisfying to the context. An optimized configuration is calculated to adapt to the current product.

Like above approaches, in [Morin 08, Morin 09b] in order to select appropriate variants, a reasoning framework treats an adaptation model to take de-

cisions. This model specifies which variants have to be selected based on a context model. Products are built by composing the selected variants (aspects) in the variability model with components in the architecture (core) model. By using the reasoning framework, a new product can be identified and adapted to current product at runtime.

Other approaches generate all members of SPL at design time in which each member is selected depending on the context. Indeed, in [Mizouni 14], based on the defined feature model, a set of members of SPL is generated by using a *SPL generator*. A member of SPL is selected according to the current context detected by using a context decision module. Similarly, Bencomo *et al.* [Bencomo 08] specify a product configuration as a variant in the variability model. Each variant is mapped to a specification of context. Corresponding to the current context, a product architecture can be selected.

In some approaches, the role of the user is mentioned to configure a software product. In CAPucine [Parra 11a], in order to build a product, a product developer selects a set of features. A desirable product is generated as a result of configuration and then deployed. In [Gomaa 07], the feature model can be configured according to given features that are selected by the user. The product line architecture is adapted and tailored to derive application architecture. The reusable components are stored in a reused library for adaptation.

Authors in [Phung-khac 10] used and extended the medium refinement process to develop the adaptation medium and generate functional medium's architectural variants. Depending on some situations of the operating environment or certain user requirements, various variants can be selected to build or adapt the running product.

Another aspect in the configuration process is reducing unnecessary features in the feature model before configuring it. In [Cetina 08a], three models are used to describe SPL, feature model, architecture model, and realization model, called SCV model. Based on a set of different evolutionary scenarios related to pervasive resources and user goals that are not feasible, the model elements in SCV can be pruned. Then, depending on the current context such as user goal, resources, an initial configuration can be generated. Other elements that are not selected for the initial configuration are integrated in the product to adapt to new context arisen at runtime. Based on the current context to build an initial product and integrating unselected elements into the initial product for adaptation are also applied in [Cetina 09a, Cetina 13].

Discussion

Existing approaches configure the variability and generate products of a family based on a given context at the configuration moment. For the given context, a set of features can be selected, and then a concrete product is built and deployed. Throughout its execution, the context can change, and the current product needs to be adapted. Usually, all features that are not selected for the initial product will be integrated into the product and deployed at runtime during adaptation process. However, integrating all features can be an issue for limited deployment targets such as mobile devices. In [Cetina 08a], such elements are eliminated thanks to pruning SCV model. For each predefined evolutionary scenario that is not feasible in the specific domain, the pruning phase applies a set of rules to delete undesired elements. However, it increases development costs and time. On the other hand, another approach is uploading required features through a network if they are available. Nevertheless, uploading required features during adaptation causes security issues as the functionality of the features may be intervened by hackers.

3.4 Supporting State Transfer

An important aspect to ensure system consistency is state transfer. Once current components are replaced by other ones, their state must be maintained and migrated into the new ones. This is a critical task to ensure correctness of the system through adaptations. This challenge is closely related to the one of ensuring consistent dynamic adaptation. In this section, we present approaches relevant to supporting state transfer.

Most existing approaches use a technique based on identifying the corresponding states between two components versions and using the getting/setting functions for transferring state. Indeed, in [Bialek 04], authors defined *state transfer function* from the current component version to the new one to represent the state mapping between two component versions. This function is manually described before performing adaptation and considered as a parameter in the adaptation request. In order to access the state of the running system, the running system has to implement the *setVar()* and *getVar()* functions, to set and get the variable values. On the other hand, a state transfer function is used in [Zhang 06] for transferring state as well.

Similarly, in [Grondin 08] an application designer defines a *state transfer net* to identify the state mapping between two component versions. This approach considers an application's state as the set encompassing values of all variable attributes of all roles in all configurations. When a component is removed from the system, component attributes tagged as variables must be saved. They are

restored afterwards to any other component replacing the current one. Information specified in the *state transfer net* allows to identify the mapping state between components versions. Each node of the net is a set of the application attributes or variables that belong to a configuration. Links of the net connect only attributes through transfer functions.

Vandewoude *et al.* in [Vandewoude 05] addressed the state transfer and proposed a methodology to deal with runtime adaptation of components. They proposed a number of steps to perform state transfer. Particularly, at design time, both, old and new component versions, are analyzed and information collected during this analysis is embedded in a structure called the *state transition logic*. This structure is packaged together with the implementation of the new component version and used by the Dynamic Update Module in which a State Transformation Manager is responsible for transferring the actual state to the new one using the state export and import actions.

Other approaches are just interested in the introspection and intercession aspects in component model. They do not take into account specifying the state mapping between two components versions, but assume that variables between two single components are similar. Indeed, the approach in [Polakovic 08] considered state as the private data encapsulated by components that are designed conforming to the Fractal component model. However, this model lacks interface for state transfer. Therefore, for the purpose of state transfer, authors extended the Fractal component model by defining additional control interfaces (*ReconfigurationController* and *StateTransferController*). The StateTransferController interface provides two services of getting and setting states. Thanks to this interface, states of component are accessible and settable.

Based also on setting/getting state with the Fractal component model, authors in [Stoicescu 12] took into account the state transfer between components based on OW2 FraSCAti v1.4 [Seinturier 12] - a platform providing runtime support for SCA and developed according to SCA principles. FraSCAti allows designers to manage stateful components which can explicitly provide a state management service in the form of a Java interface with *getState()* and *setState()* methods which is called when needed.

Similar to [Stoicescu 12], in [Chen 02], a component model was proposed in which control interface defines methods such as *extractState()* and *restoreState()* for state transfer. They are implemented by programmers and used by a configuration management in adaptation process.

In MADAM [Geihs 09] the state transfer from the old to the new configuration is also mentioned. This approach provides a simple solution in which configurable application components are implemented with interfaces that al-

low to change their states. These interfaces define methods to allow to get the current configuration state of the components, and transfer serializable one to another.

Phung *et al.* in [Phung-khac 10] used a data transfer model to represent information that the adaptation medium needs to read/write data from/to any components of the functional medium. This model is automatically created by collecting data annotation models through the generation of the functional medium variants. Based on the information specified in this model, state transfer actions can be generated in a reconfiguration plan. An adaptation manager is in charge of transferring state between component versions.

Discussion

Transferring state plays a critical role in the adaptation process to ensure system consistency. It guarantees the state of system. Many approaches are interested in this issue and propose different solutions. A simple solution is to take into consideration the homogeneity between two component versions and provide methods, setState() and getState() such as [Stoicescu 12, Chen 02, Phung-khac 10], etc. In the case of the inhomogeneity between the component versions, this solution may generate errors: type errors, data errors, etc. A solution is to propose a logic unit such as state transfer net in [Grondin 08], or state transition logic in [Vandewoude 05], that do the state mapping between the components. However, this task also remains complex if the target components have not a space for storing the state of current version.

3.5 Automatically Planning Adaptation

In order to realize the adaptation process, a reconfiguration plan should be provided. This section introduces related approaches that support planning adaptation. A reconfiguration plan can be automatically or semi-automatically generated.

One of the techniques for automatically generating a reconfiguration plan is comparing the current configuration and the new one. Particularly, in MADAM [Geihs 09], a BuilderandPlanner is responsible for dynamically building descriptions of alternative configuration versions called configuration templates. For each possible configuration version and each possible deployment, a configuration template is created. A Configurator compares the configuration templates with the information about the currently running applications to derive a sequence of steps for the adaptation process. The sequence of steps can contain

information to create, remove component instances on local or remote nodes, connect and disconnect components through local or remote connectors, and set the parameters of components.

Similarly, in DiVA [Morin 08, Morin 09b, Morin 09a], by using EMF Compare in order to compare models representing current and new versions, a *diff* and a *match* models can be produced. They specify the differences and the similarities between two versions. These models are automatically analyzed to obtain the relevant changes between the source model and the target model e.g., addition/removal of components/bindings, changes of attribute values, etc.

Also based on computing differences between two configuration versions, in CAPucine [Parra 11a], a Script Generator with two versions considered as its input is used to generate a list of modifications expressed in terms of weaving or unweaving aspects. Finally, this script is executed in the reconfiguration platform to adapt the product.

In [Pascual 14] once a new configuration fits the current context, a reconfiguration plan is generated by the Dynamic Reconfiguration Service by comparing the differences between two configuration versions. Differences are propagated to the *VSpec* tree, variation points and then the software architectural model. This plan is executed in the architecture by an adaptation service. Similarly, in [Phung-khac 10], a reconfiguration plan is generated by comparing two configuration versions. It is realized by using an adaptation medium that manages and performs the adaptation process.

Like the above approaches, in FamiWare [Gámez 11, Gámez 15], a plan is automatically generated by comparing the differences between two product configurations. This plan is interpreted by FamiWare's reconfiguration service that executes the corresponding actions in the plan.

The reconfiguration plan can be partially generated thanks to information specified at design time for adaptation. The approach in MoRE [Cetina 09a, Cetina 13] is based on the context monitor that uses the runtime state as input to check *context conditions*. These conditions are associated with predefined resolutions at design time, representing a partial configuration, e.g., a resolution has a form as {{FeatureA, Active}, {FeatureB, Inactive}}. When discovering changes in the context, and based on an appropriate resolution and current system model, a reconfiguration plan is generated by a Model-Based Reconfiguration Engine. MoRE framework is in charge of realizing actions specified in the reconfiguration plan.

In [Bencomo 08], a transition model is used to specify at a high-level abstraction the transition between variants that correspond to particular configurations

specified in the variability model. In order to adapt to a new product, the reconfiguration policies are generated from the transition model by using generative techniques.

In [Saller 13], based on the feature and context models, a transition system is derived that defines context-aware reconfiguration behavior. Every state in the system represents a valid product configuration of the DSPL that satisfies a context. This system can be extended at runtime to cope with new configurations corresponding to a particular context combination emerging during execution. The transition system provides the information necessary to execute an adaptation at runtime.

According to the approach based on analyzing features to develop adaptive software in [Lee 06], a reconfiguration request is offered by a context analyzer. Then, a reconfiguration strategy analyzer analyzes the request and determines a reconfiguration strategy. It exploits the feature model to identify reconfiguration actions in the reconfiguration strategy. Finally, this strategy is interpreted by a reconfiguration handler to realize reconfiguration actions.

Unlike the above approaches, in FUSION [Elkhodary 09, Elkhodary 10], a reconfiguration plan consists of a series of transitions from the current feature selection to a new one. Depending on a metric collected in FUSION, features impacted by a new context are considered. For each feature, a transition can be realized by using the dynamic adaptation service proposed in this approach.

Discussion

Automatically building reconfiguration plans is necessary to avoid error-prone and reduce adaptation costs. Existing approaches are based on system-specified models such as variability model, architecture model, context model, and so on, to generate reconfiguration plans. However, all of these approaches lack actions supporting for the state transfer and the system consistency specified in the reconfiguration plan.

In this thesis, we take into account the component-based adaptation. Other approaches such as [David 09, Léger 10, Buisson 15] are interested in the component-based adaptation as well as languages for specifying and realizing adaptation. However, in these approaches, the reconfiguration plan is manually implemented without automatically using models specified at design time.

3.6 Ensuring Consistent Dynamic Adaptation

This section introduces some approaches to ensure consistent dynamic adaptation. Such an adaptation must guarantee the whole system consistency, i.e., system state must be guaranteed and the correct completion of ongoing activities must be ensured. The former is related to state transfer mentioned in Section 3.4. This section focuses on the latter to find the best moment when an adaptation process can be performed.

In section 2.5.5, we have presented two criteria, *quiescence* and *tranquility*, that can be applied on single components to find the best moment for replacing them. These status definitions are largely used by most approaches related to consistent dynamic adaptation ([Gomaa 07, Polakovic 08, Pascual 14]). All of them propose different approaches or algorithms to drive components to the quiescent status.

In order to avoid the passive status of other components when considering the quiescent status of a component, some approaches are based on the principle of blocking new request messages. The approach in [Vandewoude 05] uses a technique proposed in [Vandewoude 04] that makes sure a component has completed all its actions, and the component system will temporarily block all new requests addressed to the component until it has been replaced by its new version. In [Stoicescu 12], incoming requests on replaced components must be buffered. Moreover, in order to maintain system consistency, components must be stopped in a quiescent status, i.e., when all internal processing has finished. These tasks are fulfilled by using FraSCAti and FScript [Seinturier 12]. Similarly, in [Chen 02], the proposed component framework analyzes and treats the interactions among components. It provides a component runtime environment and implements services for the dynamic reconfiguration. Particularly, the services support for managing interactions among components and blocking newly initiated invocations between the target component and other components.

The work in [Vandewoude 07] is based on the tranquil criteria. In this approach, a Live Update Extension Module (LUM) - an extension of the core DRACO system - allows a component to be replaced in a tranquil status and preserves this status for the duration of the adaptation.

Based on the tranquil status, the approach in [Ma 11, Baresi 16] proposed a new criterion, called *version-consistent*, to define when an adaptation process ensures global system consistency. It uses *future* or *past* labels to mark consequent transactions initiated by a root transaction. When the root transaction is initiated by a component and can initiate consequent transactions on other components, future label is marked on connections linked to those components.

In contrast, past labels are marked once the consequent transactions have finished. For example, a component A initiates a transaction that can cause a consequent transaction on a component B. A future label is then marked on the connection from A to B. Once this transaction has completed, a past label is marked on this connection. If all connections to the target component are marked as a past label, such component can be replaced. However, this approach is illustrated on a single component to be replaced. This assumes that the moment for replacing a component is independent from others.

The work in [Ghafari 12a] represents a connector-based approach to maintain the system consistency during adaptation of a component-based distributed system and defines the *serenity* status. In this approach, a new component version is activated as soon as receiving a reconfiguration request. Then, the current component version is informed about reconfiguration. This component notifies its connectors about changes. From this moment, new requests to these connectors will be driven to the new component version. In order to identify the serenity status, this approach uses a transaction controller to start and stop transactions. The last component that participates in a sequence of transactions must be identified at design time. It informs the transaction controller about the end of transactions. Once the transaction controller receives information from the last component, the transaction has finished and the current component version can be deleted from the system.

Discussion

Existing approaches are based on the concept of transaction to identify the safe status of components. However, most approaches take only into consideration the status of single components. They do not address the transactional dependencies among components. Authors in [Ghafari 12a] are interested in this issue. However, in a large architecture, implementing transaction controller in the approach becomes a very complex task. Moreover, it requires two versions of a component at the same time from the beginning to the end of the adaptation process. Finally, this approach is considered as an ad-hoc approach: they do not exploit information at design time for identifying the safe status for adaptation at runtime.

3.7 Summary

In chapter 1 we have introduced five challenges when developing adaptive software systems. They have been used to describe the related approaches in this chapter. A summary of these approaches are shown in Table 3.1.

In Table 3.1, the first column (Approach) indicates names such as approach name, author, or project and references of the approaches. The last five columns correspond to five challenges:

- *Modeling variability and commonality for adaptation (C1)*
- *Configuring and automatically building adaptive architecture (C2)*
- *Supporting state transfer (C3)*
- *Automatically planning adaptation (C4)*
- *Ensuring consistent dynamic adaptation (C5)*

In the table, a check mark ✓is used to indicate that the challenge in column is a main interest of the corresponding approach in the row, i.e., this challenge is managed by the approach, and a solution is proposed. A check mark between parenthesis (✓) indicates that the approach does not fully handle the challenge. A dash – indicates that the challenge is not addressed by the approach.

Firstly, to cope with C1, most existing approaches use models such as the feature model, OVM, or CVL to specify the variability and commonality. In the feature model, feature groups and optional features are considered as variation points, whereas the notion of the variation point is clearly defined in OVM and CVL. Differentiated from other approaches, each variant in OVM used in GENIE refers to a particular product architecture instead of an element or a set of elements of the software architecture. Thus, the variability used in GENIE does not represent variation points in the software architecture. Not using the feature models, MADAM is based on a particular component framework to describe a composition of component types. The variability is represented via the composition.

Secondly, most approaches address C2 to configure and automatically build adaptive software. However, according to Gomaa *et al.*, the user is responsible for manually selecting and tailoring a product from a set of existing artifacts in a reusable library. For adaptation, all elements that are not configured for the initial product are available, even if they are useless. Therefore, in MoRE, before configuring the product, some elements in the SCV model are pruned. This reduces the useless elements available in the system.

Next, we see that few approaches fully support C3. The state transfer in Chen *et al.*, Polakivic *et al.*, MADAM, Adaptive medium, Stoicscu *et al.*, is to exchange the atomic data between two components, i.e., the state is guaranteed without modifying it. However, if the state is transfered between two inhomogeneous components, it must be modified. Therefore, some approaches propose

No	Approach		C1	C2	C3	C4	C5
1	Adaptive medium	[Phung-khac 10]	✓	✓	(✓)	✓	–
2	Bialek *et al.*	[Bialek 04]	–	–	✓	–	–
3	CAPucin	[Parra 11a]	✓	✓	–	✓	–
4	Chen *et al.*	[Chen 02]	–	–	(✓)	–	(✓)
5	DarwinSPL	[Nieke 17]	✓	✓	–	–	–
6	DiVA	[Morin 08] [Morin 09a] [Morin 09b]	✓	✓	–	✓	–
7	FamiWare	[Gámez 11] [Gámez 15]	✓	✓	–	✓	–
8	FUSION	[Elkhodary 09] [Elkhodary 10]	✓	✓	–	✓	–
9	Ghafari *et al.*	[Ghafari 12a]	–	–	–	–	✓
10	GENIE	[Bencomo 08]	(✓)	✓	–	✓	–
11	Gomaa *et al.*	[Gomaa 07]	✓	(✓)	–	–	(✓)
12	HyVar	[Mauro 16]	✓	✓	–	–	–
13	Lee *et al.*	[Lee 06]	✓	–	–	✓	–
14	MADAM	[Floch 06] [Geihs 09]	(✓)	✓	(✓)	✓	–
15	MaDcAr	[Grondin 08]	–	–	✓	–	–
16	Mizouni *et al.*	[Mizouni 14]	✓	✓	–	–	–
17	MoRE	[Cetina 08a] [Cetina 13]	✓ ✓	✓ ✓	– –	– ✓	– –
18	Pascual *et al.*	[Pascual 14]	✓	✓	–	✓	(✓)
19	Polakovic *et al.*	[Polakovic 08]	–	–	(✓)	–	(✓)
20	Saller *et al.*	[Saller 13]	✓	✓	–	✓	–
21	Stoicescu *et al.*	[Stoicescu 12]	–	–	(✓)	–	(✓)
22	Trinidad *et al.*	[Trinidad 07]	✓	–	–	–	–
23	Vandewoude *et al.*	[Vandewoude 05] [Vandewoude 07]	–	–	✓	–	(✓)
24	Version consistency	[Ma 11] [Baresi 16]	–	–	–	–	(✓)
25	Zhang *et al.*	[Zhang 06]	–	–	✓	–	–

Table 3.1 – Related component-based approaches to the defined challenges

logic units, functions, or nets to map the state among components. However, all them suppose that it is necessary to have a target variable to store the state. The state will be lost if it does not exit the target variable to store it.

Similarly, most approaches supporting C1 are interested in C4. For C4, the feature models are exploited to plan the adaptation. Based on a new configuration meeting new requirements, reconfigurations actions can be identified.

Finally, most approaches do not fully handle C5. They are based on the quiescent or tranquil criteria to find the best moment for adaptation. However, as previously mentioned, these criteria only consider single component without the transactional dependency for the system consistency. The approach proposed by Ghafari *et al.* takes into account this issue. It can be applied on reconfiguring a components group. However, this is an ad-hoc approach.

Limitations of the Related Approaches

From the description of related approaches in the five above sections, we see that existing limitations consist of the lack of:

1. **A process to guide engineers on how to specify variability and commonality.** Indeed, existing approaches use feature model, OVM, or CVL to specify the variability and commonality of adaptive systems. However, no existing guideline helps engineers on how to specify them. In addition, separation of variation points in the variability specification is necessary to clearly identify changes in the architecture for adaptation at runtime.
2. **Separation of useful/unuseful elements for building adaptive systems.** To cope with resources limitations and security issues during adaptation, the distinction of useful/unuseful elements is necessary. The unuseful elements should be eliminated from the system. However, most approaches assume that all architectural elements are available at runtime for adaptation. Or, if such elements are not available, they must be available over the network. However, in terms of security, the transmission of software elements during the software execution should be avoided.
3. **An abstraction of specifying state transfer for adaptation.** State transfer is a complex task. In existing approaches this task is manually implemented or represented in a form interpreted by a particular reconfiguration framework.
4. **Necessary actions in an auto-generated reconfiguration plan for adaptation.** Indeed, in existing approaches, a reconfiguration plan focuses on component replacement actions without actions for driving replaced components to a safe status, and transferring state among compo-

nents.

5. **Specification information that is exploited at runtime to ensure dynamic consistent adaptation.** Most existing approaches are based on managing transactions to detect the safe status. Usually, a particular mechanism is used to manage the transactions. Additionally, the safe status is considered in single components without transactional dependencies between them. And, no existing approaches address managing transactional dependencies among components for adaptation.

In order to overcome these limitations, the two research questions identified in Section 1.3 should be answered. On one hand, for the question *How to build adaptive software architectures?*, we work on a development process to build adaptive software architectures. Such a development process provides the necessary guidelines to specify variability, and architecture models of an adaptive system. In addition to specify variability and software architecture, a configuration specification of an adaptive product in the development process can provide necessary information to identify useful/useless elements in the adaptive architecture. Moreover, the role of participants in the process needs to be determined.

On the other hand, for the second question, *What is the appropriate moment to start the execution of an adaptation process?*, we work on an adaptation mechanism that are based on managing transactions to identify the best moment for adaptation. Such mechanism is built based on exploiting information specified at design time in the variability model. This information is used at runtime to support for generating reconfiguration actions. Additionally, to provide the information for state transfer, a state transfer model is specified at design time that is also exploited at runtime to identified the state transfer actions in the reconfiguration plan.

Conclusion

We have presented the related approaches to develop adaptive software systems. Based on the five challenges identified, we have discussed various aspects in those approaches. From each considered aspect, limitations are identified that allow to justify the research questions to be important that need to be answered. To deal with such questions, the next part presents our contribution in which two chapters focus on responding these two questions.

Part II

Contribution

Chapter 4

Adaptive Software Architecture Development Process

Contents

4.1 Chapter Overview

In Chapter 1, we have introduced two research questions of this thesis, *How to build adaptive software architectures?*; *What is the appropriate moment to start the execution of an adaptation process?*. This chapter addresses the first question by proposing a development process with concrete activities in which variability modeling is a critical one. A variability model is specified and used throughout the process. When it is configured, an adaptive software architecture can be generated. Our approach is based on CVL to specify variability and generate the adaptive software architecture. CVL offers tools and meta-models to specify variability of a product family but it does not offer a method to specify the variability model and the base model. In addition, the product is generated using CVL with no runtime variability. Therefore, considering CVL and related approaches leads us to two main questions:

- *Question 1:* How to specify variability and base models?
  This question relates to a subprocess to specify them. Such a subprocess orders activities and relations of them.
- *Question 2:* How to configure the variability model and generate an adaptive software architecture that contains only the necessary elements for adaptation?
  By default, an adaptive software architecture contains all architectural elements identified in the base model. As previously mentioned, our approach aims at eliminating unnecessary elements from the final architecture.

In order to answer these questions, this chapter aims at defining a development process to build adaptive software architectures that include only the elements that may be needed in the target environment. Such a development process must provide concrete steps with corresponding guides for engineers to specify the variability and the base models. Moreover, the role of participants in such process must be defined. To do that, Section 4.2 firstly presents the adaptive software architecture development process. Section 4.3 focuses on domain engineering - a subprocess in the development process in which modeling variability is a main task. Next, Section 4.4 presents how to configure the variability model and steps on how to generate adaptive software architectures. Furthermore, a general adaptive software architecture is shown to see the structure of an adaptive system. Finally, Section 4.5 presents implemented tools and a case study to validate our approach.

4.2 A Global Overview of Development Process

Figure 4.1 shows our adaptive software architecture development process. It consists of different steps performed from high-level specification to executable code. Our process is based on SPL engineering. The SPL engineering distinguishes two phases: domain engineering and application engineering.

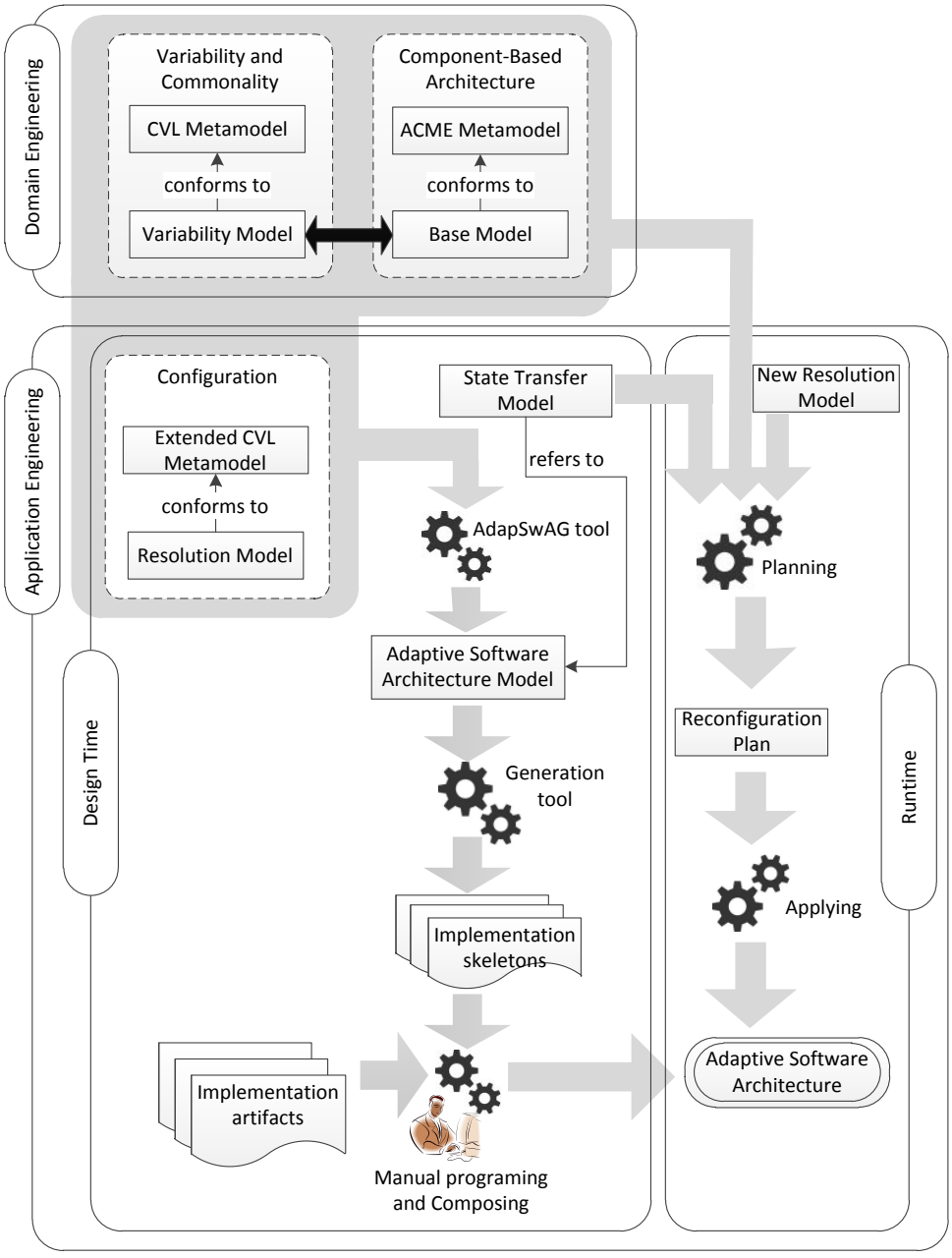
The top of the figure describes the domain engineering process. It is responsible for defining the commonality and variability, and architecture of a product line. The variability and the base models are artifacts of this phase. In our approach, the variability model is specified using CVL, whereas, the base model is represented by using ACME. We have chosen ACME as it offers generality enough to straightforwardly describe a variety of system structures. Besides, it provides a good basis for designing and manipulating architectural specifications and generating code.

The bottom of the figure describes the application engineering process. It refers to the process of actually combining the artifacts obtained during the domain engineering phase to generate ***adaptive software architectures***, also called ***adaptive products***. In our approach, this phase is partitioned into two subprocesses: one at design time and another one at runtime. The former, described in the left part of the figure, focuses on configuring the variability model, generating ***adaptive software architecture models***, also called ***adaptive product models***[1] that include variability, and then executable code to build particular adaptive products that can be deployed in a target platform, e.g., OSGi. The latter, in the right part of the figure, describes an adaptation process at runtime in which a reconfiguration plan is executed by a *Reconfigurator* that is injected into the software architecture to control the adaptation process.

4.3 Domain Engineering

In this section, the whole process of domain engineering described in the top of Figure 4.1 is detailed. As previously mentioned, this process focuses on specifying variability and base models. To do that, domain issues are identified and analyzed by engineers. Roles of stakeholders in this process must be distinguished from collecting information in the domain, analyzing, and modeling it. We propose three scenarios to analyze and specify these models. The scenarios

1. The terms of {adaptive software architecture and adaptive product}, {adaptive software architecture model and adaptive product model} are used as synonyms in the remainder of this dissertation.



AdapSwAG tool: **Adap**tive **S**oft**w**are **A**rchitecture **G**eneration tool

Figure 4.1 – Adaptive software architecture development process

provide a guideline that consists of various steps to specify the models.

4.3.1 Roles

We consider different roles for software engineers and participants in the domain engineering. Based on artifacts needed to be defined in this process, we distinguish three roles:

- Domain expert: Domain experts have a high degree of skills in their knowledge of the domain. In the development process, they provide necessary knowledges for domain engineers to identify what is the variability and commonality of the domain.
- Domain engineer: Domain engineers are responsible for collecting information/data from the domain, and the domain expert. They define variability models.
- Domain architect: Domain architects are architectural experts of the domain. They perfectly know about architecture of the domain. Together with domain engineers, they can participate in collecting information from domain and designing systems. In particular, they are in charge of describing base models.

4.3.2 Variability Modeling Process

In the domain engineering, the variability modeling plays an important role. This section presents strategies to specify variability models conforming to the CVL meta-model, and base models conforming to an architecture meta-model, i.e., ACME meta-model in our approach. Such strategies order the following set of activities:

- **Activity 1:** Identifying the variability and commonality in the domain, and design information.
- **Activity 2:** Specifying the *VSpec* tree and constraints in the tree.
- **Activity 3:** Building the base model.
- **Activity 4:** Mapping the *VSpec* tree and the base model.

These activities closely relate with each other. The first activity is the basis to realize the second and third activities. The result of the second activity

can give the necessary information to realize the third activity and vice versa. The last activity is realized according to the result of the second and the third activities. Figure 4.2 shows the activities and their sequencing in time.

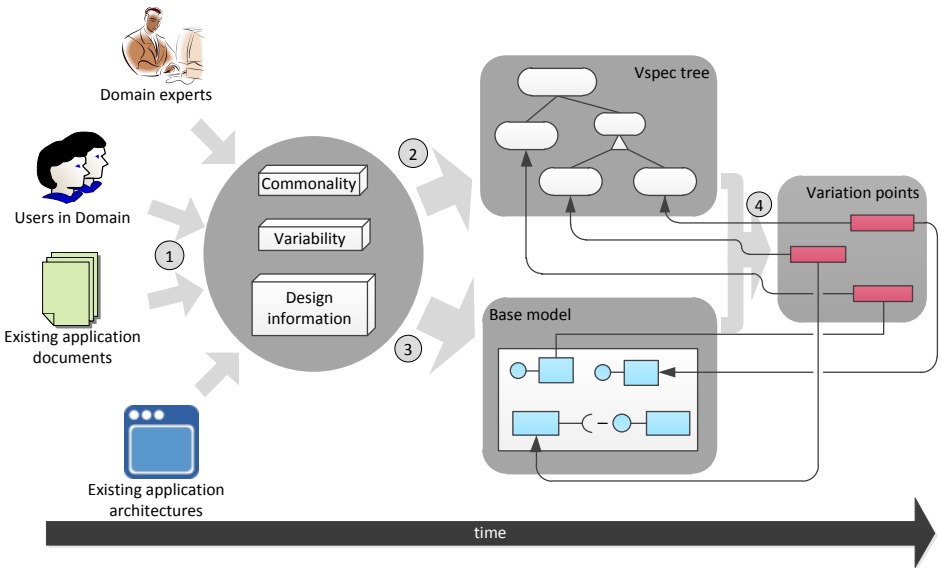


Figure 4.2 – Four activities in a variability modeling process

The first activity, ① in the figure, aims at identifying variability and commonality in the domain. This activity is performed by domain engineers who analyze documentation of existing applications of a product line, interview domain experts or users of the domain, etc.

Once the commonality and variability are identified, they are organized into a *VSpec* tree as the result from the second activity ②. This activity is manually realized by domain engineers. The *VSpec* tree will be used throughout the process to manage variability of the product line and configure particular products. On the other hand, based on the information collected from the first activity, constraints among *VSpecs* that can not be presented in the hierarchic structure of the tree can be identified. Then, these constraints are represented by using excludes/implies constraints and integrated into the variability model.

The third activity ③ aims at specifying a base model. Depending on the applied strategy, it is defined before or after specifying the *VSpec* tree. This activity is performed by domain architects.

Finally, variation points are defined in the activity ④ to map *VSpecs* in the *VSpec* tree and elements in the base model. This activity can be manu-

ally realized by domain engineers or semi-automatically done thanks to similar characteristics of elements in the two models.

Depending on the order of activities ② and ③, three different strategies may be identified for a development process. We discuss them in the rest of this section.

"Variability-driven process" (top-down approach)

Figure 4.3 shows the development process using this strategy. The *VSpec* tree is specified from information collected by domain engineers before specifying the base model. Following this strategy, a base model may be built based on the results from the first and the second activities. It allows specifying variability at high level of abstraction towards the diversity of components at concrete level.

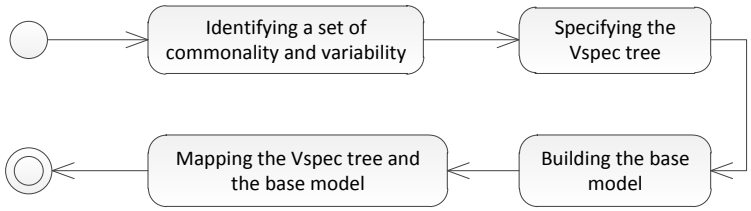


Figure 4.3 – "Variability-driven process" strategy

This strategy facilitates the building of the base model. For example, the approach in [Phung-khac 10] allows to generate architectural variants from the variability model via a refinement process. Each architecture variant is considered as a configuration of the base model. Based on the variability model to build the base model ensures the matching between elements in the variability and the base models. This strategy may be interesting when building an adaptive product in a new domain where there are no existing products, or reusing existing products whose architecture models are not explicit.

"Architecture-driven process" (bottom-up approach)

Figure 4.4 shows the development process using this strategy. Activity ② needs the result from the first activity as it is difficult or even not possible to only use the base model to identify the variability. This strategy allows specifying diversity of components at concrete level towards high level abstraction.

This strategy is appropriate for reusing architectural models of existing products in a domain that is explicit. These architecture models can be directly

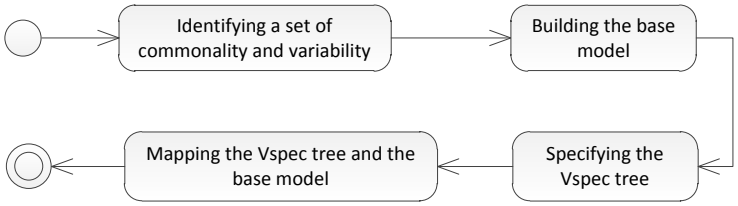


Figure 4.4 – "Architecture-driven process" strategy

reused to create the base model of a product line by merging them (a technique to merge models is mentioned in [Rubin 13]). Reusing the architectural models to build the base model allows to reduce the time as well as the development costs. However, identifying the variability in the base model may be a complex task.

"*VSpec* tree - base model independent process" (hybrid approach)

Figure 4.5 depicts this strategy. The *VSpec* tree and the base model are independently specified. The advantage of this strategy is to allow independently specifying the models, i.e., they can be parallelly developed. This reduces the development time. Unlike the two first strategies, there is no guarantee to have a variation point for each *VSpec*. For example, building the variability and the base models from particular product models in [Martinez 15a, Martinez 15b] could be considered as a "*VSpec* tree - base model independent process".

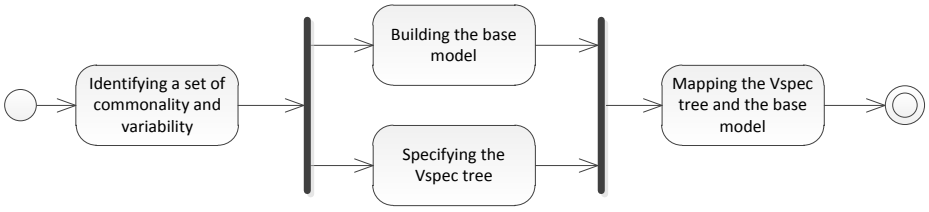


Figure 4.5 – "*VSpec* tree - base model independent process" strategy

This strategy is appropriate for reusing artifacts of heterogeneous products in a domain. There may be existing architecture models in heterogeneous forms. Therefore, it should be uniformed before merging them. Moreover, the *VSpec* tree can be built without the base model.

4.3.3 Summary

In our development process, domain engineering encompasses all the activities of the variability modeling process. The goal of the domain engineering process is to build the variability and the base models that will be used throughout the application engineering process to build adaptive software architectures.

In this section, we have distinguished the different roles of stakeholders to specialize the tasks in the domain engineering process. Specializing the tasks increases the productivity and the effectivity of the work in the domain engineering process.

On the other hand, we have defined three scenarios on how to specify the variability and the base models. They provide an order of the activities to build them. According to practical conditions of the availability of the collected information, a corresponding strategy could be selected. For example, if the architectural variants exist in the same domain, the second strategy should be selected to build a base model in short time. An answer to *Question 1* has been given in this section.

4.4 Application Engineering

This section presents a process to build particular products from information specified in the models in the domain engineering. In traditional SPL, such a product is generated without its variability. In our approach, a product must include its variability, i.e., it contains alternative elements for adaptation. Thus, configuration of variability must provide the necessary information to identify such elements. Moreover, in order to control adaptation at runtime, a controller unit should be added in the architecture.

Application engineering focuses on various activities to generate an adaptive architecture and control adaptation at runtime. Therefore, the roles in this process should be identified as well.

4.4.1 Roles

The responsibilities for building adaptive software architectures are shared among several stakeholders:

- User: Users in the application engineering directly use the product resulting from the application engineering. They provide the necessary require-

ments to build particular products from artifacts in the domain engineering. Such requirements can be met by reusing existing artifacts in the domain engineering. They may contain information about the variability for building the adaptive products.

- Adaptive product designer: Adaptive product designers collect requirements provided by the users. Based on such requirements, they define a resolution model to configure the variability model defined in the domain engineering. They use generation tools to generate the adaptive product model from the specified models, and then part of the executable code.
- Product developer: Product developers complete the executable code generated from the process, compose, and package it.
- Adaptation engineer: Adaptation engineers are responsible for developing adaptation controllers.

4.4.2 Design time

In general, at design time, an adaptive product should be created from models in the domain engineering. It contains only the necessary elements for adaptation. To do that, our process at design time starts with specifying a resolution model. As the adaptive product should be able to adapt at runtime, this model should allow adaptive product designers to specify which variability has to be effectively managed at runtime.

4.4.2.1 Configuring Variability: Extensions of CVL

As previously mentioned, our approach is based on CVL. An extract of the CVL resolution meta-model [OMG 12] is presented in Figure 4.6 (right part). A resolution model is used to configure a variability model conforming to the variability meta-model (left part of Figure 4.6) in which each *VSpec* is resolved by a *VSpecResolution*.

When building an adaptive software architecture, the *Choice VSpec* plays an important role. A *Choice VSpec* allows to choose one or several of the possible alternatives, i.e., components in the architecture may be different depending on the chosen alternatives. A *Choice* is resolved by given a value to the *decision* attribute of the corresponding *ChoiceResolution*. If a *Choice VSpec* is resolved by a TRUE value for this attribute, the corresponding elements in the base model will be present and activated in the adaptive product.

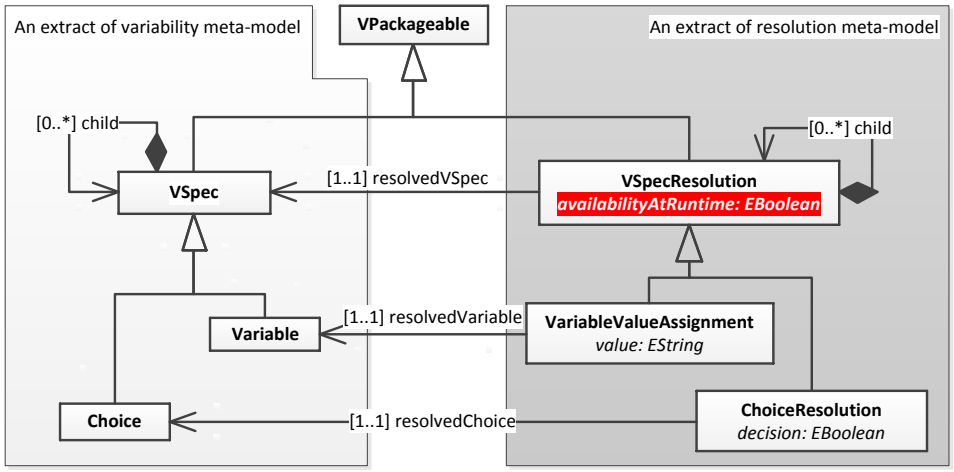


Figure 4.6 – An extract of CVL meta-models

Although the information about the active elements is important, it is not enough to build an adaptive software architecture. Indeed, such architecture may include [2] elements that are not activated in the initial product, but needed if the execution context or use requirements change.

To specify such elements, we have added a new attribute to a *VSpecResolution*, ***availabilityAtRuntime***. This attribute indicates if the corresponding *VSpec* should be included or not in the adaptive product at runtime.

Attributes		***availabilityAtRuntime***	
		true	false
decision	true	present and activated	present and activated
	false	present and not activated	not present and not activated

Table 4.1 – Configuring elements in the product based on the *decision* and ***availabilityAtRuntime*** attributes

Table 4.1 shows the association of the *decision* and ***availabilityAtRuntime*** attributes for configuring variability and generating an adaptive software architecture model. As previously mentioned, a *ChoiceResolution* with the *decision* attribute set to TRUE indicates that its corresponding elements in the

2. We use this term to indicate that the architecture effectively contains the elements or that there are some mechanisms available to add them "on-the-fly".

architecture are present and activated in the adaptive product. When a TRUE value is assigned to the *decision* attribute, the role of the ***availabilityAtRuntime*** attribute need not to be taken into account. On the other hand, when the *decision* attribute value is FALSE, the availability of components in the product at runtime depends on the value assigned to the ***availabilityAtRuntime*** attribute. If the value is TRUE, its corresponding elements in the architecture should be present in the product at runtime and thus available in case of adaptation. Otherwise, the corresponding elements are not included in the product.

4.4.2.2 Adaptive Product Model Generation

Figure 4.7 shows the stage of our development process where the variability model is configured and the adaptive product model generated. As already mentioned, the adaptive product model is generated by configuring the variability model based on the resolution model. This task is realized by the AdapSwAG tool. It considers the CVL model (the variability model, the base model and the extended resolution model) as its input. Its output is an adaptive product model that may include alternative elements for adaptation. Compared to the CVL execution of the CVL approach, the software architecture generated by AdapSwAG tool may include some variabilities. The AdapSwAG tool generates this adaptive product model by applying the following rules:

1. A component in the base model will be included in the adaptive product model and activated in the adaptive product if the *decision* attribute corresponding *Choice VSpec* is set to TRUE.
2. A component in the base model will be present in the adaptive product model, but not activated in the adaptive product if the corresponding *VSpecResolution decision* attribute is set to FALSE and the ***availabilityAtRuntime*** to TRUE.
3. Value for variables in components are obtained from the *value* attribute of their *VSpecResolution*.
4. Connections between components that should be present in the adaptive product model and activated in the adaptive product are maintained.

The generated adaptive product model is independent of the target platform. Based on the adaptive product model and the given target platform, a text generation module generates implementation artifacts skeleton that include packages. Each of them corresponds to a component specified in the adaptive product model and consists of component implementations, a component specification, and configuration files. Once the implementation artifacts skeleton is

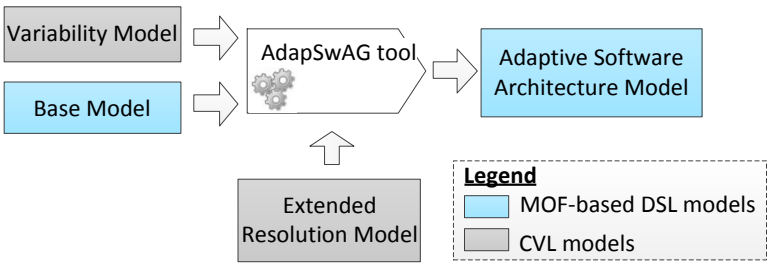


Figure 4.7 – Generating Adaptive software architecture model (adapted from [OMG 12])

generated, implementation artifacts that are either available or should be developed are integrated into it to build components. This task is performed by product developers. The result is an adaptive product that can be executed in the target platform.

On the other hand, if the adaptive product model embeds variability, then, its architecture may change at runtime, and the component state affected by the changes should be migrated to the new one. As the semantic of this state is component-specific, the state migration task can not be completely automated. Thus, adaptation engineers have to specify a state transfer model that gives actions to effectively migrate the state between components. The state transfer model represents the state mapping between placement and replacement components [3] in the adaptive product model. It is specified at design time and used at runtime. The state transfer model will be detailed in Chapter 5.

4.4.2.3 Adaptive Software Architecture

In order to realize adaptation at runtime, the generated adaptive product needs to have a control unit, or adaptation controller. Figure 4.8 shows the general adaptive architecture that we consider. The adaptive product is considered as a managed system that is deployed in a reflective component platform, e.g., OSGi. It can be implemented according to a component model, e.g., iPOJO or Fractal. On the other hand, the adaptation controller is considered as a managing system to manage and control adaptation.

In the figure, the adaptation controller has three main parts: *Repository*, *Reconfigurator*, and *Planner*. These parts closely interact with each other in the adaptation process.

3. In our approach, we use placement component to indicate the current component version that will be replaced by another one called replacement component

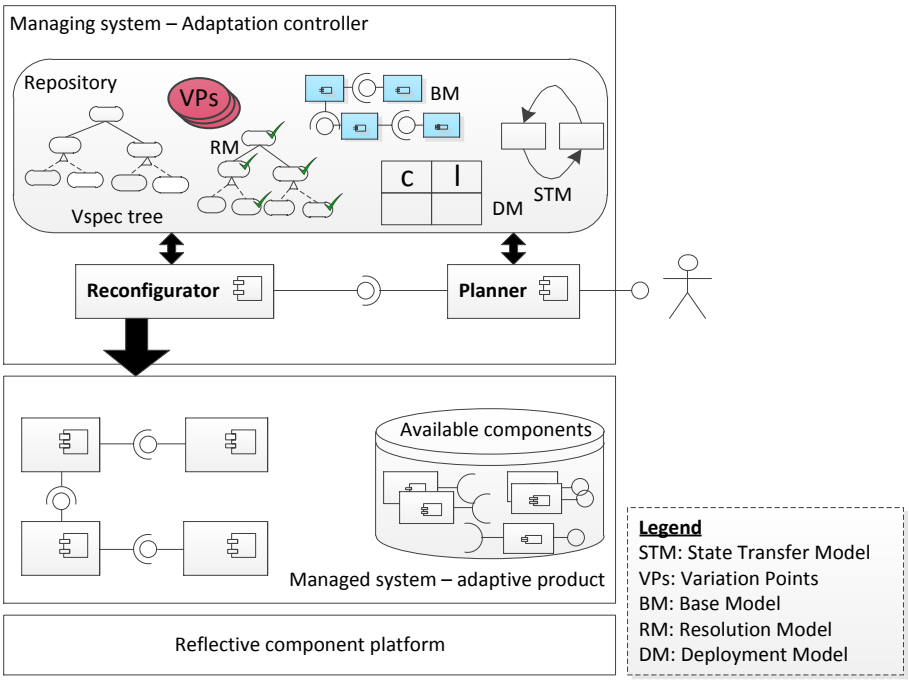


Figure 4.8 – General adaptive software architecture

Repository

It contains models specified in the domain engineering and the application engineering at design time such as the CVL model, the state transfer model, and the deployment model. The latter is used to get information about the location of components in the system. It is out of the scope of this thesis.

The repository corresponds to the *Knowledge* part in the MAPE-K model. It contains information to be exploited by the *Planner* and the *Reconfigurator* for generating reconfiguration plans and effectively realizing adaptation, respectively.

Planner

It is in charge of generating reconfiguration plans after receiving an adaptation request with a new resolution model. To do that, it exploits information in the repository. A generated plan is injected into the *Reconfigurator* to realize the adaptation process.

Reconfigurator

It is responsible for executing actions in the reconfiguration plan received from the *Planner*. The *Reconfigurator* reads the plan and coordinates actions specified in the plan. Actions are executed using introspection or intercession services of the target platform. In addition, it can use control services of components to invoke particular services. A component model describing these services will be presented in Chapter 5.

4.4.3 Runtime

Figure 4.9 details an adaptation process. When the software architecture has to be changed, a new resolution model (new configuration) must be specified. This configuration may be either defined by engineers or computed thanks to a control module. This activity is represented by step ① in the figure.

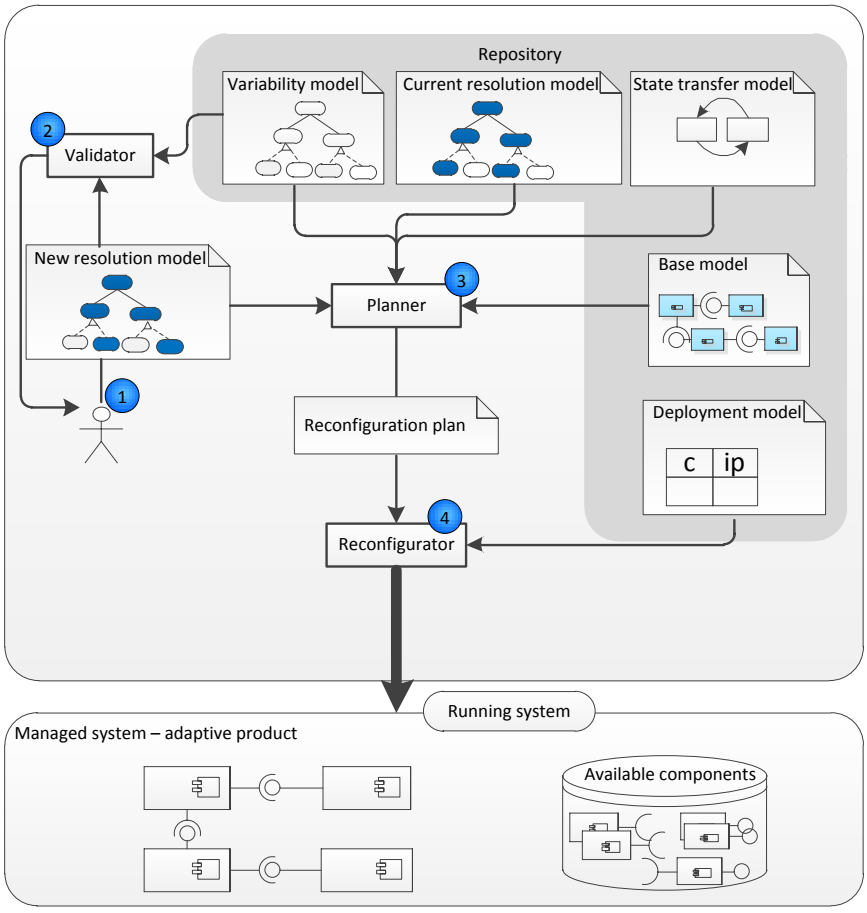


Figure 4.9 – An overview of adaptation process

The consistency of the new resolution model is validated by a validation tool (step ② in the figure). If the new resolution model does not conform to the variability model, the validation tool informs the adaptation engineer to correct the new resolution model. Otherwise, it is sent to the *Planner* to plan adaptation.

The *Planner* compares the current resolution model in the repository and the new one to identify what *VSpecs*, elements and connections in the architecture will be selected or unselected (step ③ in the figure). For state transfer actions, it uses the state transfer model available in the repository.

In step ④, the reconfiguration plan is sent to the *Reconfigurator* that reads it and executes the corresponding actions. It makes an extensive use of the models in the repository. Details of this utilization will be given in Chapter 5.

An important problem is to determine when components replacement actions can be started while preserving system consistency. As previously mentioned, this issue relates to transaction management. It will be detailed in Chapter 5 and a solution will be presented.

4.4.4 Summary

In this section, we have addressed *Question 2* of our research. We have introduced a subprocess to build adaptive software architectures from the specification of models in the domain engineering. The main goal of this subprocess is to build adaptive software architectures that include only elements that may be needed in case of adaptation. To do that, a new attribute ***availabilityAtRuntime*** has been proposed. It allows adaptive product engineers to determine what elements have to be available at runtime for adaptation. Eliminating useless elements in the product economizes deployment space. Although proposed in the CVL context, the ***availabilityAtRuntime*** attribute can be applied to whatever feature model to specify the need for availability of the elements in adaptive products.

In addition, we have structured an adaptive architecture in two parts, the managing and the managed systems. This architecture is based on the MAPE-K model, an adaptation model that is largely used by the adaptive software community. Therefore, it can be simply used and implemented by adaptive software engineers.

4.5 Evaluation and Discussion

In order to validate the solution proposed in this chapter, this section presents a tool that is used in the development process to build adaptive software architectures. It is used and applied on a prototype to illustrate the feasibility of our approach.

4.5.1 AdapSwAG tool: a Plug-in for Generating Adaptive Software Architectures

The Eclipse platform is built from various Eclipse components that are called plug-ins. It provides a strong mechanism to extend it by integrating additional functionalities via such plug-ins that are implemented as bundles of code. Via the additional plug-ins, we can create new software development environments as editors, transformation, generation tools, etc.

In order to support generating the adaptive product model, and the executable code in the adaptive software architecture, we have created an Eclipse plug-in, called **Adap**tive **S**oft**w**are **A**rchitecture **G**eneration tool (AdapSwAG tool). It consists of three modules to validate resolution models, generate the adaptive product model from the models specified in the domain engineering, and executable code from the adaptive product model. These modules are integrated into the Eclipse platform as new menu entities (see Figure 4.10).

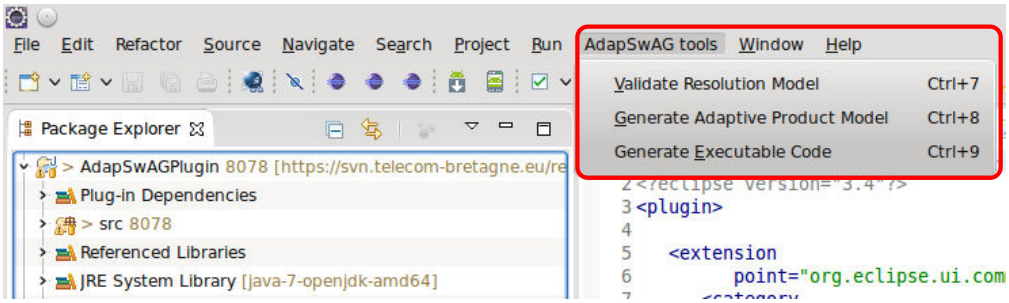


Figure 4.10 – AdapSwAG tool integrated into Eclipse platform

4.5.1.1 Validation of Resolution Models

This module allows validating a resolution model to be applied to successfully configure a variability model. It analyzes constraints, hierarchic relations, and cross-over constraints via *implies* or *excludes* constraints in the variability model.

Without loss of generality, we suppose that the variability model is structured as a tree by using *Features*. In order to configure this model, a resolution model is specified as a tree structure. Each element of the resolution model is a *FeatureResolution* that maps to a *Feature* in the variability model. Suppose that a *FeatureResolution* has a *decision* attribute to represent the selection of the corresponding feature, and an ***availabilityAtRuntime*** attribute to represent the availability of that *Feature* in the product configuration.

We defined an algorithm to validate if a resolution model satisfies the constraints specified in a variability model. It is based on the following definitions:

- $F = \{F_1, F_2, ..., F_n\}$ denotes a set of *Features* of the variability model.
- $FR = \{FR_1, FR_2, ..., FR_m\}$ denotes a set of *FeatureResolutions* of the resolution model.
- $E = \{(F_i, F_j) \in F \times F : F_i \ excludes \ F_j\}$ denotes a set of excludes constraints of the variability model
- $I = \{(F_i, F_j) \in F \times F : F_i \ implies \ F_j\}$ denotes a set of implies constraints of the variability model
- FR_i denotes the *FeatureResolution* corresponding to F_i
- $FR_i.decision$ represents the *decision* on F_i. If the $FR_i.decision = true$, F_i will be selected (or positively resolved) and its corresponding elements in the base model will be present and activated in the adaptive product, and vice versa.
- $FC = FR_i.getChildren()$ represents children of FR_i in a resolution model.

Algorithm 1 summarizes the constraint analysis process to validate the resolution model. It takes as inputs: 1) the variability model; 2) the resolution model; 3) the set of constraints. Lines 4-9 analyzes mandatory constraints to ensure that a *Feature* must be positively resolved, if its parent is positively resolved. Lines 10-21 are used to analyze the *cardinality* constraints. This constraint identifies the number of *Features* children that must be positively resolved in a group. Finally, lines 25-36 analyze the *implies/excludes* constraints satisfaction in the resolution model.

4.5.1.2 Product Model and Code Generation

The validated resolution model is used as an input to generate the adaptive product model. For each *FeatureResolution* in the resolution model, the corre-

Algorithm 1 Validating the resolution model

```
1: procedure VALIDATION(FR, F, E, I)
2:     for each FR_i ∈ FR do                          ▷ Search the resolution model
3:         Fi ← FR_i.getReferenceFeature
4:         if FR_i.decision = false then
5:             if Fi.isMandatory AND
6:                 FR_i.parent.decision = true then
7:                 return false
8:             end if
9:         else
10:            if F_i has cardinality then
11:                min ← F_i.getLower()
12:                max ← F_i.getUpper()
13:                {FC} ← FR_i.getChilds()
14:                for each FC_k ∈ FC do
15:                    if FC_k.decision = true then
16:                        count++
17:                    end if
18:                end for
19:                if count < min OR count > max then
20:                    return false
21:                end if
22:            end if
23:        end if
24:    end for
25:    for each e(F_i, F_j) ∈ E do                    ▷ Check excludes constraint
26:        if FR_i.decision = true
27:            AND FR_j.decision = true then
28:            return false
29:        end if
30:    end for
31:    for each ι(F_i, F_j) ∈ I do                    ▷ Check implies constraint
32:        if FR_i.decision = true
33:            AND FR_j.decision = false then
34:            return false
35:        end if
36:    end for
37:    return true
38: end procedure
```

sponding *Feature* can be identified, and the elements in the base model identified via the mapping between the variability and the base models.

Adaptive product model generation is shown in Algorithm 2. It is based on the set of rules described in Section 4.4.2.2. In the algorithm:

- $C = \{C_1, C_2, ..., C_l\}$ denotes a set of components
- $CN = \{(C_i, C_j) \in C \times C\}$ denotes a set of connections from component C_i to component C_j
- $BM = \{C, CN\}$ denotes the base model

Algorithm 2 Generating the adaptive product model

```
1: procedure ProductModelGeneration(FR, F, BM)
2:     PM = {}
3:     for each FR_i ∈ FR do
4:         if FR_i.decision = true then
5:             F_i ← FR_i.getReferenceFeature        ▷ Get the corresponding
   Feature from the FeatureResolution
6:             C_l ← F_i.getReferenceElement         ▷ Get the corresponding
   elements in the base model
7:             PM.add(C_l)
8:         end if
9:     end for
10:    for each CN_i(C_m, C_n) ∈ Cn do
11:        if C_m ∈ PM AND C_n ∈ PM then PM.add(CN)
12:        end if
13:    end for
14:    for each FR_i ∈ FR do
15:        if FR_i.availabilityAtRuntime = true AND
16:            FR_i.decision = false then
17:            F_i ← FR_i.getReferenceFeature
18:            C_l ← F_i.getReferenceElement
19:            PM.add(C_l)
20:        end if
21:    end for
22:    return PM
23: end procedure
```

In Algorithm 2, lines 3-9 generate the adaptive product model whose components correspond to *Features* that are positively resolved. Connections among such components will be guaranteed in the adaptive product model (line 10 -

13). Otherwise, for the *Features* that are negatively resolved but that should be available at runtime, the corresponding components will be present in the adaptive product model but not activated. Thus, lines 14-21 add such components to the adaptive product model.

The adaptive product model generated is used as input to generate the execution code. In our approach the code generation module is implemented using the Xpand generator framework [4]. This generator is based on EMF, i.e., it generates code from EMF models, the adaptive product model in our case. XPand provides a statically-typed template language that allows us to write code generation templates. The templates teach the code generator how to translate the adaptive product model into code.

Based on the given execution platform and the structure of the target component model, a code generation template can be written. An extract of this template is shown as Figure 4.11. This template is used to create Java classes each corresponding to a component specified in the adaptive product model. Line 2 allows to create Java files with its ≪name≫. From line 3, the content of Java files is described.

In order to run the Xpand generator, a workflow needs to be defined. The workflow controls generation process whose steps (loading models, checking them, generating code) are executed by the generator. In our implementation, the code generator is integrated into the AdapSwAG tool. Depending on the target platform, different file types will be automatically generated.

In our approach, we use iPOJO/OSGi [Escoffier 07] as a target platform. The result of code generation contains the necessary elements such as Java implementations, a component specification (XML documents), and a configuration file to create iPOJO components.

4.5.2 A case study

4.5.2.1 Example description

In order to illustrate our approach, we have studied a Medical Image Diagnosis System (MIDS) that includes three main elements: equipments (e.g., X-ray CT scanner and X-ray imaging devices) that take medical images, an image store (IS) that stores the medical images sent from the equipments, and a tablet that receives the images from the server treated by a doctor (see Figure 4.12). A Wifi or 3G connection may be used to connect the tablet to the server. The tablet

4. http://wiki.eclipse.org/Xpand

```
1 «DEFINE component FOR Component»
2     «FILE name.toLowerCase()+"/"+name+".java"»
3 package «name.toLowerCase()»;
4 «EXPAND importBinding FOR this»
5 «EXPAND importBindingImpl FOR this»
6 «EXPAND importAttachment FOR this»
7 «EXPAND importAttachmentImpl FOR this»
8 import org.apache.felix.ipojo.annotations.*;
9 @Component(name="«name»")
10 /*
11  * Begin the class
12  */
13 public class «name»
14     «EXPAND implementAttachment FOR this»
15     «EXPAND implementsBinding FOR this»
16 {
17     «EXPAND property FOREACH getProperties()»
18     «EXPAND binding FOR this»
19     «EXPAND attachment FOR this»
20 }
21      «ENDFILE»
22     «IF representations.size > 0»
23         «FOREACH getRepresentations() AS representation»
24             «IF representation.getSystems().size > 0»
25                 «FOREACH representation.systems AS system»
26                     «IF system.getComponentList().size > 0»
27                         «EXPAND component FOREACH system.getComponentList()»
28                     «ENDIF»
29                 «ENDFOREACH»
30             «ENDIF»
31         «ENDFOREACH»
32     «ENDIF»
33 «ENDDEFINE»
```

Figure 4.11 – An extract of a Xpand template file

can use a buffer to store medical images. When connection is strong, tablet memory may be released allowing the system to work online, i.e., the tablet does not need to use the buffer to store the medical images. Additionally, other services can be used to observe and evaluate the system such as a log service, a performance evaluation service, etc.

Studying the MIDS leads us to consider a simple client-server system in which the tablet plays the role of a server that receives images from IS. Images are treated by a doctor and analysis results returned to the IS. Other elements of the MIDS play the role of a client that sends images, considered as requests, to the tablet and waits the analysis results considered as responses from the tablet. In this section, we use the “server” and “client” terms to indicate the tablet and the IS, respectively. Furthermore, we consider the images sent by IS as client messages.

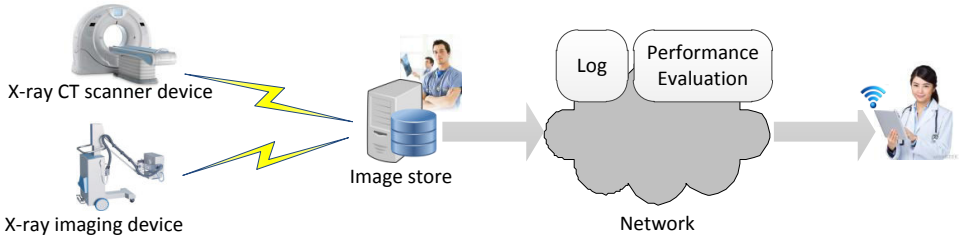


Figure 4.12 – A medical image diagnosis system

Figure 4.13 shows the models and tools used at design time to build an adaptive software architecture for the MIDS. The left side of the figure presents the CVL models, the right side the stages to generate the product. As previously mentioned, the server can work online or with a buffer to store the images. Therefore, we consider two types of servers: a buffered server (`BS`) and a non-buffered server (`nBS`) (see the variability model in Figure 4.13). Client messages treated by servers can be logged by a log server (`Log`). We also assume that the initial configuration of the application includes `Client` and `BS`, and that the `Log` server is not necessary. As engineers want to reduce the size of the server when network connection is strong, the product should be able to replace the `BS` by the `nBS` at runtime.

4.5.2.2 Specification and Implementation

Variability of this application is modeled by a *VSpec* tree in CVL that conforms to the CVL meta-model. All *VSpecs* in our *VSpec* tree are *Choices*. The solid lines indicate mandatory *VSpecs*, and the dotted lines optional *VSpecs*. The [1..1] multiplicity value associated with a small triangle indicates that only one *VSpec* should be configured [Haugen 13].

In this example, there are five variation points in the variability model. An *ObjectExistence* variation point does the mapping between the `Client` *VSpec* in the *VSpec* tree and the `Client` component in the base model. This point indicates that the `Client` component may or may not exist in the product. Similarly, four *ObjectExistence* variation points do the mapping between `Server`, `nBS`, `BS`, `Log` in the *VSpec* tree, and `Server`, `nBS`, `BS`, and `Log` components in the base model, respectively. Moreover, the multiplicity value in the *VSpec* tree indicates that one and only one server, either `BS` or `nBS`, can be configured in `Server`.

In order to generate a product, the *decision* attribute of the `Log` and `nBS` nodes of the resolution model is set to FALSE to indicate that they should not be

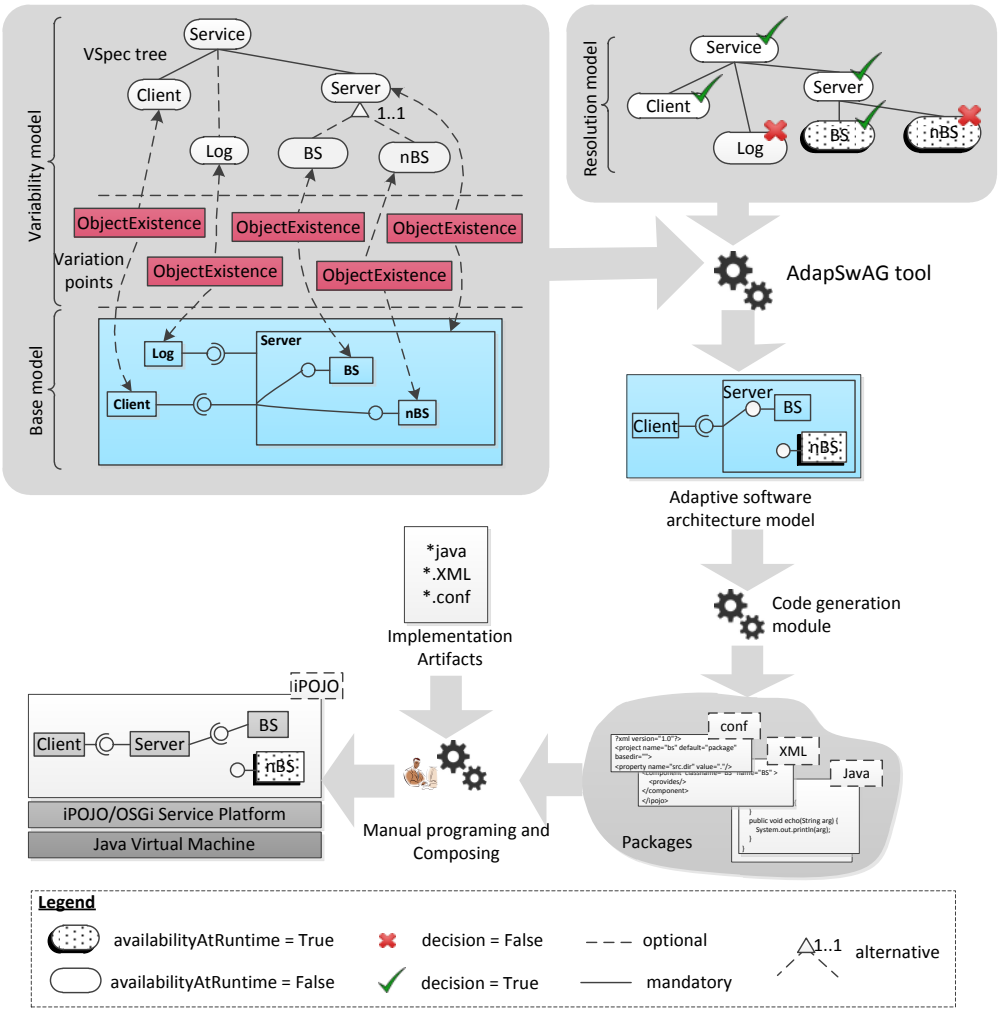


Figure 4.13 – An example of building an adaptive software architecture

activated in the initial product. However, as the ***availabilityAtRuntime*** attribute of the `nBS` is set to TRUE, the corresponding components will be present in the product for a later activation. On the other hand, the ***availabilityAtRuntime*** attribute of the `Log` *VSpecResolution* is set to FALSE to indicate that the corresponding elements will not be present in the product. Hence, the ***availabilityAtRuntime*** attribute allows to eliminate the unnecessary components (e.g., the `Log` component).

The resolution model must conform constraints specified in the *VSpec* tree. One of these constraints is that the `Server` component can be implemented either by a `nBS` or a `BS` one, i.e, both `nBS` and `BS` cannot have the *decision* attribute set to TRUE. Such validations ensure the validity of the resolution model. In our approach, they are validated by a module integrated in the AdapSwAG tool.

AdapSwAG tool generates an adaptive product model with an adaptive software architecture that no longer contains the `Log` component (the top right part in Figure 4.13). Compared to the CVL execution element described in Section 2.3, the product model in the CVL approach would not contain both components, `Log` and `nBS`.

In this example, we use the iPOJO/OSGi platform [Escoffier 07] as a target platform since it allows to build dynamically extensible Java-based applications and facilitates management including features like dynamic dependency, component reconfiguration, component factory, and introspection. Furthermore, it also works on many Java virtual machines such as BEA JRockit, Mika, or Google Dalvik (used in Android) [Escoffier 08]. In order to connect the components, the Apache CXF framework [5] is used.

Based on the specification of the iPOJO component model and Apache CXF framework, a text generation module is implemented with the Xpand generator framework [6]. It generates implementation artifacts skeletons that include packages. Each generated package corresponds to a component specified in the adaptive product model. It consists of the necessary elements such as Java implementations, a component specification (XML documents), and a configuration file to create iPOJO components.

Although not a real example, this experimentation shows that our approach is feasible using existing technologies: iPOJO, CVL, ACME, CXF. We used the top-down approach to build the variability and the base models as we had no existing architecture model. The adaptive product includes two versions of the server component as we decided that both of them may be useful at runtime. The `Log` component was pruned to reduce the code volume.

4.5.3 Discussion

In this chapter, we have introduced an adaptive software architecture process to build adaptive software architectures. This process consists of various steps to specify variability and generate the adaptive software architecture.

5. http://cxf.apache.org/
6. http://wiki.eclipse.org/Xpand

In addition to specify the variability and the base models, they must be configured to generate the adaptive product model, and the adaptive product that can be executed in a target platform. A generated adaptive product model should only contain necessary elements for adaptation. Therefore, we have proposed a new attribute, ***availabilityAtRuntime*** in the resolution model. This attribute allows to select elements that are not configured in the initial product to be included in the adaptive product model.

Moreover, a generation process has been proposed to generate the adaptive product. It consists of two steps: generating the adaptive product model, and the executable code. We have implemented a tool that is in charge of the generation process. This tool exploits models specified at design time to generate the adaptive product model, and is based on iPOJO/OSGi for code generation. Furthermore, we have provided general algorithms applied in the tool to perform generation. These algorithms can be implemented for other models and target platforms. As each target platform assumes a particular component model, the generation module should take it into account. We present the properties of the component model so that adaptive software architectures can be generated in the next chapter.

In order to control the adaptation process, an adaptation controller or a managing system should be developed and composed with the adaptive product. Next chapter details our proposed adaptation process that ensures the consistency of the adaptive product.

Chapter 5

Consistent Dynamic Adaptation Process

Contents

5.1 Chapter Overview

This chapter addresses a main issue to ensure system consistency before and after adaptation: the correct completion of ongoing activities and the validity of the new version of the adaptive product. This issue relates to transaction management that allows to identify when transactions start and finish in the system, i.e., transaction scope. A transaction is a sequence of actions executed by one or several components that completes in a bounded time.

Our approach is based on specifying the transaction scope at design time to build adaptive products. Then, the transaction scope specification is exploited at runtime to identify the best moment to realize components replacement. Therefore, considering transaction management leads us to putting out the following research questions:

- *Question 3:* How to specify and manage transactions scope in adaptive software architectures?

  Transactions scope allows to identify the transactional space to be managed when changes have to be made in the architecture. A management module that exploits transactions scope specification can be developed and integrated into the product. It is in charge of determining when transactions start and finish.

Additionally, an adaptation process relying on transaction management should be proposed. The process should ensure system consistency, i.e., all ongoing activities have correctly completed, and the state must be guaranteed before and after the adaptation. The following question should be answered.

- *Question 4:* How to identify when to replace components and realize state transfer for consistent dynamic adaptation?

  Identifying the best moment to replace components plays a critical role as it will influence the correctness of ongoing activities. Moreover, the state integrity should be taken into account. An adaptation process that ensures the correct completion of ongoing activities and guarantees the system state is called *consistent dynamic adaptation* process in this thesis.

Section 5.2 describes how to specify transaction scope and how it is exploited for building adaptive software architectures and to manage transactions at runtime. Next, adaptation strategies based on the proposed transaction management are presented in Section 5.3. The main activity in those strategies is isolating the set of components to be replaced. Section 5.4 will detail how an

adaptation process takes place. Finally, Section 5.5 provides evaluation and discussion of this chapter.

5.2 Transaction Management

This section details how transaction scope is specified in our approach and how it is exploited to build adaptive products and manage transactions at runtime so that ongoing activities complete correctly. To clarify the transaction management issue, a case study is described and applied throughout this chapter.

5.2.1 A Case Study

5.2.1.1 Example Description and Needs of Transaction Management

The case study used in this chapter is adapted from the one presented in Figure 4.12 (Chapter 4). Images exchanged in the system are coarse, i.e., they are not processed before being sent/received. In some systems such as ones in defense that require a high level of performance (e.g., defense domain), the exchanged images should be compressed. Therefore, two main functionalities, compression and decompression, are added into the original system. The images will be compressed before being sent to the receiver, and be decompressed after arriving at the receiver. The system can be re-described as in Figure 5.1. The sender exploits the images in the image source, compresses, packages, and sends them to the receiver (steps ①, ②, ③ and ④ in the figure). After receiving the images and decompressing them, the receiver exports them to the screen (steps ⑤ and ⑥ in the figure). The compression/decompression functionalities can be implemented using different algorithms, each presenting a different level of performance. Depending on the different performance requirements, the implementation of such functionalities may be changed at runtime.

Without loss of generality in the adaptation context, the software architecture of the system can be described by an Architecture Description Language (ADL), and the images in the communication process can be considered as communication messages (see Figure 5.2). The figure presents a graphical representation of the architecture of the image delivery system. It consists of five main components, namely `Sender`, `Receiver`, `Compression`, `Decompression`, and `Packer`. In the figure, two implementations of the compression and decompression functionalities have been considered with two compression/decompression algorithms, Run-Length Encoding (RLE) [Watson 03] and Lempel-Ziv (LZ) [Farach 98]. The considered scenario is shown in Figure 5.3. Whenever a

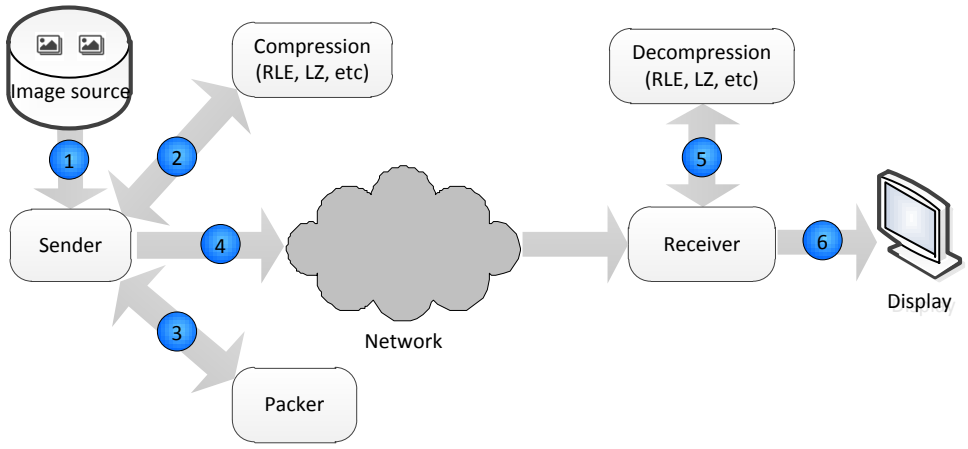


Figure 5.1 – The image delivery system example

message is ready, it is sent by the `Sender` component to the `Compression` one. Then, the compressed message is sent to the `Packer` that adds a header to the message. When the complete package is ready, it is sent to the `Receiver` component. As soon as the `Receiver` component receives the package, the message that is extracted from the package will be sent to the `Decompression` one to be decompressed.

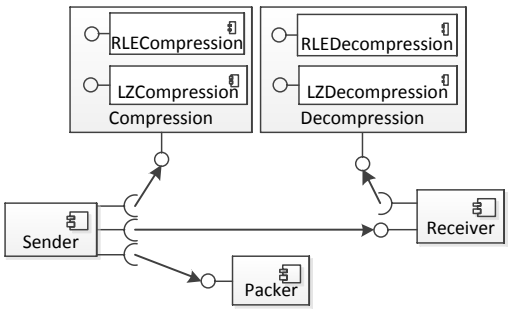


Figure 5.2 – Adaptive software architecture of the image delivery system

The adaptation process considered in this chapter simply consists of the replacement of the algorithm used by the `Compression` component by another one. In this case, the `Decompression` component should also be replaced to use the corresponding decompression algorithm. If we replace the `Compression` and `Decompression` components at runtime, we should ensure that all messages that have been compressed before replacement are decompressed using the right algorithm, i.e., the adaptive software architecture should guarantee the global consistency.

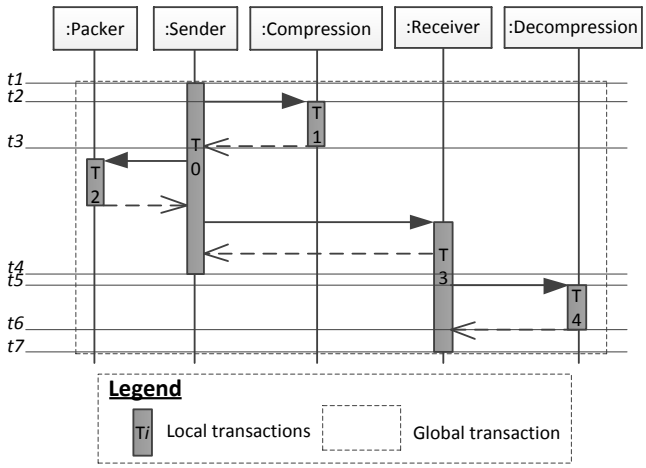


Figure 5.3 – Behavior of the image delivery system

According to the *quiescent* criterion, when considering the single `Compression` component, it reaches quiescent status before t1 or after t4. For the `Decompression` component, it reaches the quiescent status before t1 or after t7. Therefore, when considering both components, `Compression` and `Decompression` replacement can only be made before t1 or after t7 in Figure 5.3. If the `Sender` always initiates new transaction before the end of the transaction T3, the `Compression` and `Decompression` components will never reach the quiescent status. Therefore, **all** components that can directly or indirectly initiate a new transaction on this component must be in passive status (defined in [Kramer 90]). Consequently, quiescence often causes significant disruption to the running system.

In order to reduce the disruption, Vandewoude et al. [Vandewoude 07] proposed the concept of *tranquility*, as an alternative to *quiescence*. The `Compression` component reaches the tranquil status before t2 or after t3. Similarly, the `Decompression` component reaches the tranquil status before t5 or after t6. Therefore, when considering the tranquil status of two components at the same time, *tranquility* criterion allows to replace them before t2, from t3 to t5, or after t6. Moreover, the replacement of the `Compression` and `Decompression` components are independent because they are not adjacent components. However, changes from t3 to t5 on the system do not ensure system consistency, i.e., the images may be compressed by an old version of the algorithm and decompressed using the new one. This leads to incorrect decompressed images compared with the original ones, i.e., global consistency is not guaranteed. Replacement of `Compression` and `Decompression` components after t3 and before

t5 should then be forbidden.

We assume that network conditions ensure response transfer from the `Decompression` component to the `Receiver` one. Between t2 and t3, t5 and t6, the `Compression` and `Decompression` components cannot be replaced as they are actively processing a request or engaged in servicing a transaction. After t6 (after the end of transaction T4), images compressed by the `Compression` component have already been decompressed by the `Decompression` component. Therefore, replacing the `Compression` and `Decompression` components after t6 guarantees global consistency. In terms of transactions, if T1 happens, T4 must be completed before replacing components. This means that there is a dependency between transactions T1 and T4, that we call *dynamic dependency*, or *transactional dependency*. The `Compression` and `Decompression` components cannot be replaced after the beginning of T1 and before the end of T4.

Hence, identifying such dependencies between transactions is necessary. To this end, we distinguish two transaction types, *local* and *global transactions*.

Definition 5. ***(Local transaction)*** *A local transaction is defined as a sequence of actions executed by a single component that completes in bounded time. It can be initiated by the component itself or another one.*

Definition 6. ***(Global transaction)*** *A global transaction is a set of local transactions to realize a complete task. It can be initiated and finished at the same component or another one.*

According to these definitions, if there are dependencies between local transactions on components in the same global transaction, they must complete their role in the global transaction before replacing them. In our approach, dependencies between local transactions are specified at design time and exploited at runtime to identify when to replace components while ensuring global consistency.

5.2.1.2 CVL Specification

Figure 5.4 shows a *VSpec* tree, a base model (as a component-based architecture), and variation points for our case study. The resolution model is omitted in the figure for the sake of clarity.

In order to model variability in the system, we only use *Choice VSpecs* and represent them by rounded rectangles. Each *VSpec* can be bound to its parent by a solid or a dotted line. A solid line means that if the parent is positively resolved, the *VSpec* child has to be positively resolved. A dotted line means

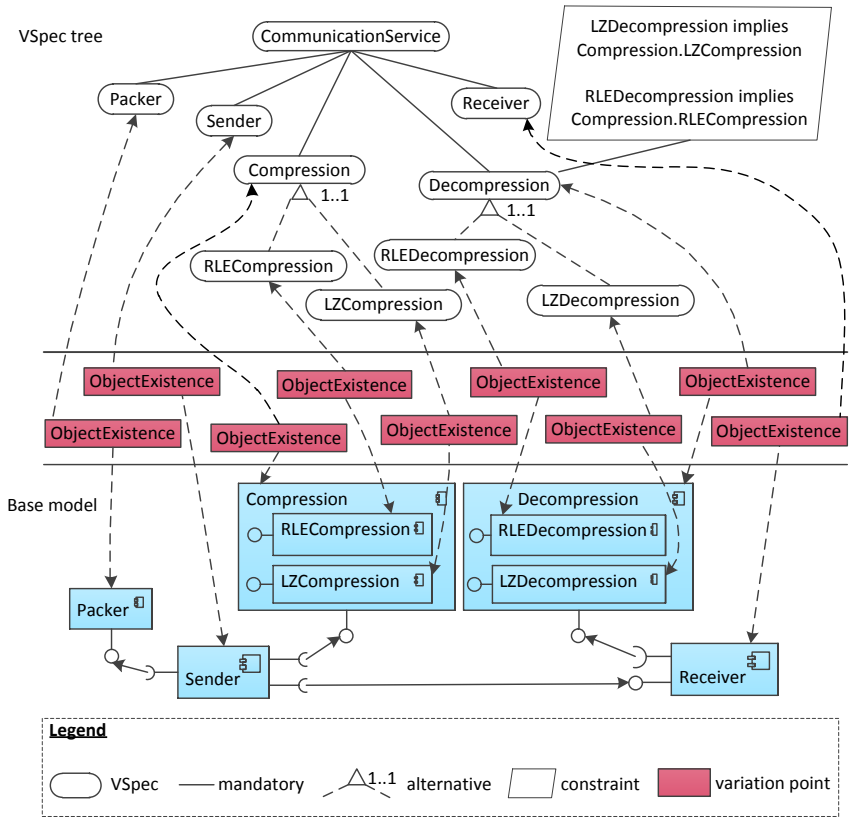


Figure 5.4 – Modeling variability using CVL

that if the parent is positively resolved (chosen), the *VSpec* child is optional (chosen or not). For instance, if the `Compression` *VSpec* is positively resolved, it is not necessary to choose `RLECompression` or `LZCompression`. However, the [1..1] multiplicity value associated with a small triangle in the figure indicates that if the parent is positively resolved, one *VSpec* child must also be. In this case, either `RLECompression` or `LZCompression` must be positively resolved in the configuration.

Variation points link *VSpecs* with elements in the base model. We use the *ObjectExistence* variation point in our example. An *ObjectExistence* variation point is bound to the `RLECompression` *VSpec* and the `RLECompression` component, which means that, if the `RLECompression` *VSpec* is negatively resolved (not chosen), the `RLECompression` component will be removed from the resulting configuration.

5.2.2 Transactional Dependency Specification in CVL

OCL constraints may be specified in CVL between *VSpecs*. For example, a constraint like `A` *implies* `B` means that if `A` is positively resolved and its corresponding elements in the base model are included in the architecture, then `B` should also be positively resolved and the corresponding elements in the base model also included in it.

Such dependencies are not enough when building adaptive software. For example, a smart home may include an alarm service (AS) and a presence detection service (PS). One may want to specify that if the PS is included in the product, the AS should also be included, i.e., PS *implies* AS. However, if PS is replaced from an IR-based motion sensor system to a thermal-based one, the AS does not need to be replaced to keep the smart home consistent. Additionally, when replacing the IR-based motion sensor by the thermal-based sensor, information treated by the IR-based motion sensor need not be taken into account by AS. Hence, there is no transactional dependency between AS and PS. On the contrary, when replacing the `Compression` component by another version, messages already compressed by the `Compression` component must be decompressed by the `Decompression` component before realizing the adaptation. Therefore, the *implies* or *excludes* constraints are not enough when building adaptive software.

To tackle this issue, we propose a new constraint, ***dependsOn***. A constraint like `A` ***dependsOn*** `B` means that when replacing `A`, messages in a transaction where `A` is involved must also be processed by `B` before replacing the components `A` and `B`.

Number	Constraint
#1	RLECompression ***dependsOn*** RLEDecompression
#2	LZCompression ***dependsOn*** LZDecompression

Table 5.1 – An extract of new constraints in our approach

The ***dependsOn*** constraint may be used to decide when to perform components replacement. For instance, a message compressed by the `RLECompression` component must be decompressed by the `RLEDecompression` one before replacement. This means that, both must be configured at the same time in the system configuration. Therefore, the `RLECompression` *VSpec* must imply the `RLEDecompression` *VSpec* in the variability model (see Figure 5.4), and vice versa. Moreover, when replacing the `RLECompression` and `RLEDecompression` components during execution, messages compressed by `RLECompression` must

be decompressed by `RLEDecompression` (see Figure 5.3), i.e., both must finish their role in the global transaction. Constraints presented in Table 5.1 are used to specify transactional dependencies between the `RLECompression` and `RLEDecompression` components. Similarly, there is a transactional dependency between `LZCompression` and `LZDecompression`. The direction of the ***dependsOn*** constraint is explained later in Section 5.2.3.

5.2.3 Transactional Dependency Management

As mentioned in Section 5.2.1.1, the components to be replaced must finish their role in all current global transactions before replacement. Therefore, we need to identify the beginning and the end of global transactions in the base model. In this section, we distinguish different roles of components for this purpose. Additionally, a *Transaction Manager* needs to be added into the architecture. It communicates with other components by using the control functions of the component model.

5.2.3.1 The role of components

In component-based software architecture adaptation, an adaptation process realizes the necessary actions to replace a set of components, called ***placement components group***, by another one, called ***replacement components group***. In our approach, we distinguish two types of placement components groups (see Figure 5.5).

- A placement components group without ***dependsOn*** constraints consists of components in which a component can require or not services of other components in the group to complete its role in the global transaction (Figure 5.5a).
- A placement components group with ***dependsOn*** constraints consists of a set of components in which there is at least one transactional dependency specified in the variability model (the dotted arrows in Figure 5.5b). In order to complete a global transaction in the group, the transaction also requires services of components outside the group. This is presented by the dotted line in the figure.

In order to replace all components in a placement components group, the group should be isolated from the rest of the system [1]. Its components must have

1. Based on an idea in the surgery domain, a surgeon needs to clamp the organ to be removed or replaced.

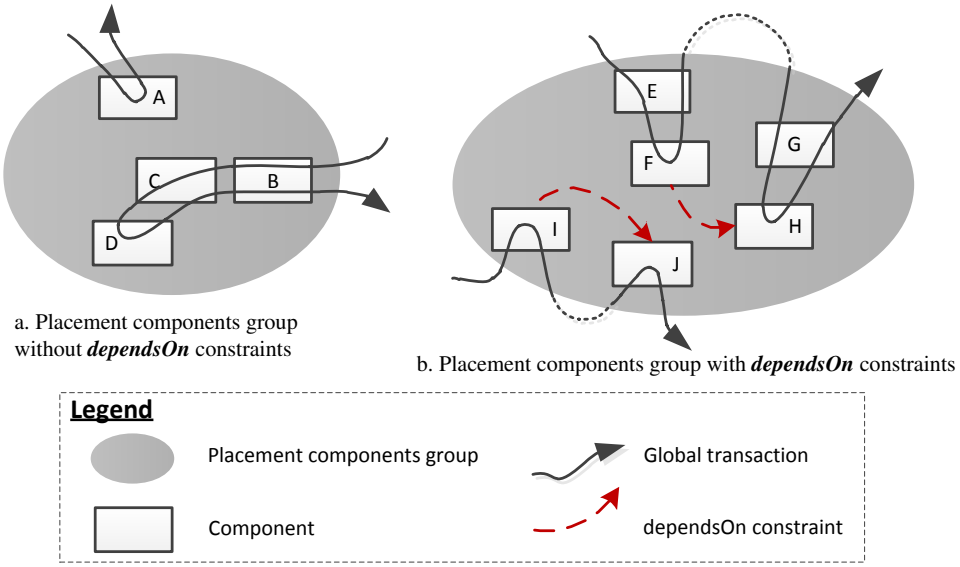


Figure 5.5 – Two types of placement components groups

finish their role in all current global transactions in the group before replacing them. Isolating a placement components group is performed in our approach by activating a barrier in frontier components in the group. Therefore, the ***frontier*** and ***internal components*** should be differentiated. Frontier components provide services for external components, e.g., components `A, B, E, G, I, J` in Figure 5.5. Internal components have no relation with components outside the group, e.g., components `C, D, F, H` in Figure 5.5. Frontier and internal components can be identified by using the base model, once the placement components group has been identified. This will be detailed in Section 5.4.2.1.

Once isolated, replacement will be possible when no component in the group is engaged in servicing a global transaction and transactional dependencies are managed. In oder to know when these conditions apply:

1. All components manage their status information and make it available. Two values are possible for this information:

   Definition 7. ***(Busy status)*** *The busy status indicates that the component is treating a request or engaged in servicing a transaction.*

   Definition 8. ***(Free status)*** *The free status of a component indicates that the component is passive and not currently engaged in servicing a transaction.*

All components in the group should be free before replacement.

2. Components with a transactional dependency make available information about completed global transactions. Such components can be identified by the ***dependsOn*** constraint in the specification. We call them ***dependent components***. For example in Figure 5.5b, components {`F` and `H`}, and {`I` and `J`} are dependent components. To better explain the role and responsibilities of dependent components, we call the source component of the constraint the ***starter component*** (e.g., `F`, `I`), and the destination component of the constraint the ***ender component***, (e.g., `H`, `J`).

Once a starter component has participated in a global transaction, it must be processed by the ender component before allowing the replacement of such components. To this end, a starter component should be able to indicate for each global transaction in the group, if it already processed the transaction. For each global transaction in the group already executed by the starter component, an ender component should be able to tag the transaction as finished by the dependent components.

In our approach, starter and ender components should be designed with the necessary control functions to manage global transactions:

- Starter components generate global transaction control information (component name, transaction ID) and add it to communication messages.
- Ender components remove global transaction control information after treating them.

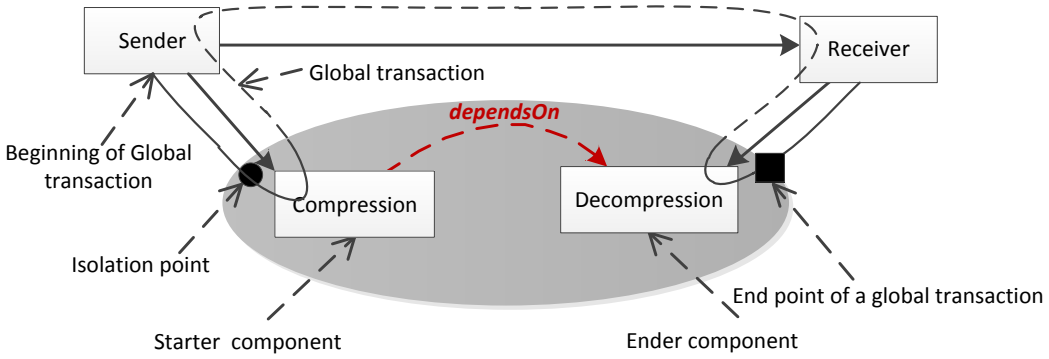


Figure 5.6 – A placement components group with ***dependsOn*** constraint

Figure 5.6 shows a global transaction in our example (described in Section 5.2.1.1). It corresponds to a placement components group with a ***dependsOn***

constraint. `Compression` and `Decompression` components are both frontier and dependent components (starter and ender, respectively). As frontier components, they should include a barrier to isolate the group. As dependent components they should manage global transaction control information in communication messages.

5.2.3.2 Transactions Manager and Reconfigurator

As presented in Chapter 4, in order to change an adaptive software architecture at runtime, an adaptation controller controls the adaptation process. We have identified three main parts of the controller, one of them, the *Reconfigurator*, controls the execution of changes. It decides about when components replacement can start and control its different steps. To start a component replacement process:

- All internal and frontier components should be *free*, and
- Dependent components, if any, should already finish to process all current global transactions in the group.

The *Reconfigurator* gets the first type of information from the components themselves: the status information. For the latter type of information, a particular component, the *Transaction Manager*, is added to the general adaptive software architecture at design time. Based on transactional dependencies, dependent components are connected to the *Transaction Manager* and send it information about when a global transaction has been processed by the starter component and when it has been finished by the ender one.

Figure 5.7 shows the final general adaptive software architecture of our example. `Compression` and `Decompression` components are connected to the *Transaction Manager* as they are dependent components. They are also connected to the *Reconfigurator* for realizing control functions during adaptation. This will be detailed in Section 5.2.3.3.

Figure 5.8 shows messages exchanges among a general placement components group, the *Transaction Manager*, and the *Reconfigurator*. Starter and ender components inform the *Transaction Manager* about the beginning and the end of global transactions, respectively. Once the *Reconfigurator* wants to start components replacement, it asks the *Transaction Manager* if dependent components have finished to process current global transactions. If so, the *Reconfigurator* waits for other components to be *free*.

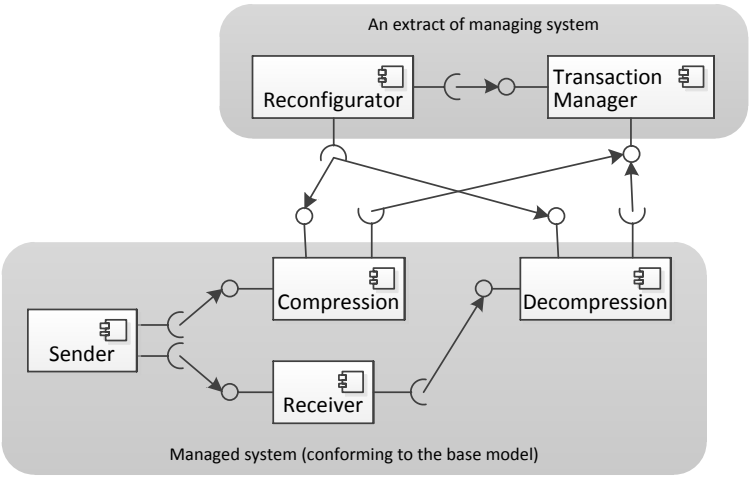


Figure 5.7 – A final general adaptive software architecture of the example

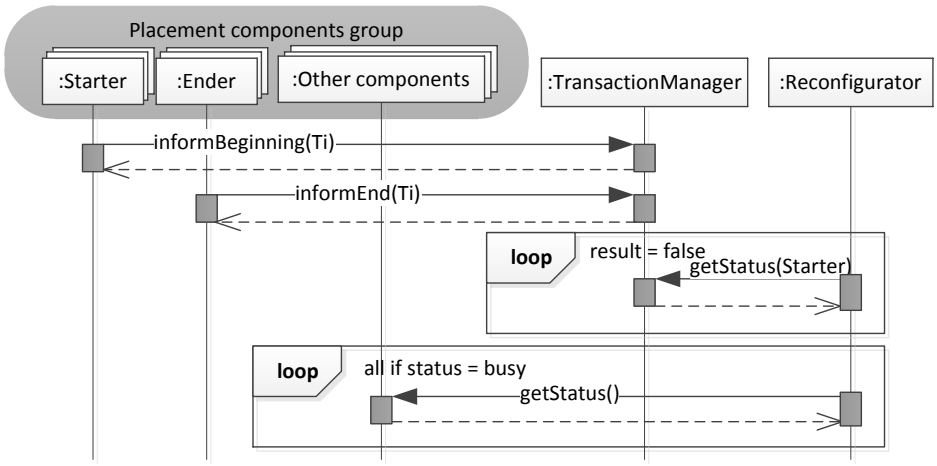


Figure 5.8 – General interaction of components in a placement components group, the *Transaction Manager*, and the *Reconfigurator*

5.2.3.3 A Component Model for Consistent Adaptation

As previously mentioned, components should offer a set of control functions so that:

- Frontier components should be able to isolate the placement components group.
- Starter and ender components should be able to inform the *Transaction Manager* about the beginning and the end of global transaction process-

ing.

- All components should manage a status information and make it available.

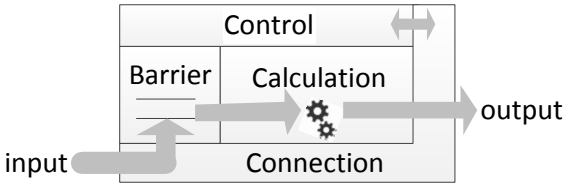


Figure 5.9 – Component model

To this end, we propose a component model presented in Figure 5.9. It consists of four parts: *Barrier*, *Calculation*, *Connection*, and *Control*. The barrier is used to isolate placement components groups. The calculation is the functional part of the component model. A component can require or offer services from/to other ones via the connection. The control part is in charge of executing actions received from the *Reconfigurator* component. It can activate/deactivate the barrier and process the transactional control information received from starter components in the group.

Control

The control part of our component model provides control services that the *Reconfigurator* can invoke to manage adaptation process. We have identified eight control services:

- Isolation/Reintegration services: they allow the *Reconfigurator* to indicate a component to activate/deactivate its barrier to isolate a placement components group.
- Passivation: it allows the *Reconfigurator* to tell a component to passivate so that it does not start new transactions anymore.
- Connection redirection: it is used by the *Reconfigurator* to change functional connections of components.
- State transfer: it allows the *Reconfigurator* to read and write the component state information.
- Status service: it allows the *Reconfigurator* to get the status (*busy* or *free*) of components.
- Finally, the control part includes functionalities to process the global transaction control information, and inform the *Transaction Manager*

about the beginning and the end of the global transactions in dependent components

In order to use the control services on components, the *Reconfigurator* needs to be connected to them via the connection part of the component model.

Connection

There are two types of connections: functional connections and control connections. Functional connections correspond to connections for application functionalities. The communication messages in this type of connections are service requests to perform the functionalities of the system. Control connections are used to connect components to the *Reconfigurator* or the *Transaction Manager* components. The exchanged messages in this kind connections are control messages, state of components, or global transaction control information.

Barrier

The barrier of a component plays a critical role in our approach. It is activated by the control part in the component model in frontier components. Such a barrier may block all types of messages or not depending on the adaptation strategies. These strategies and the rules applied by barriers are detailed in Section 5.3.2.

5.2.4 Summary

In this section, we have introduced the need of 1) specifying transactional dependencies for managing global transactions in a placement components group and 2) a component model that supports consistent adaptation.

Specifying transactional dependencies in the variability model allows to reduce the design space and facilitates exploiting them at runtime. In our approach, transactional dependencies are specified in a variability model conformed to the CVL meta-model. However, they can be adapted to other models.

Additionally, we have distinguished two types of placement components groups to limit the number of components that need to be managed in terms of transactions by the *Transaction Manager*. Only dependent components need to be managed by the *Transaction Manager*. To support consistent adaptation, we proposed a new component model that clearly identifies the control functions for adaptation, and explicits the need for a barrier. Our component model can be implemented on many component-based frameworks, e.g., Fractal, iPOJO,

to build adaptive software architectures.

Transactional dependencies and our component model give an answer to *Question 3* "How to specify and manage transaction scope in adaptive software architectures?". The next section gives details on how the components barrier may behave and the consequences of this behavior on the adaptation strategy.

5.3 Adaptation Strategies Based on the Isolation of the Placement Components Group

5.3.1 Two Adaptation Strategies

Depending on the filtering rules applied by the components barrier of a placement components group, we distinguish two adaptation strategies: *hard isolation adaptation strategy* and *soft isolation adaptation strategy*.

Hard isolation adaptation strategy (HIAS): In this strategy no filtering rules are applied by the barrier. All messages arriving to components of a placement components group are blocked and stored by the barrier. Components replacements starts as soon as the group is isolated, but the running transactions in the group may not be completed. Once the replacement finishes, messages stored in barriers should be unblocked to be processed by new components, so that uncompleted transactions are "replayed".

As the placement components group is isolated from the rest of the system, component replacement may begin immediately and there is no need for a *Transaction Manager* component. However, a complex store-and-replay process or rollback mechanism is necessary to ensure the global consistency.

Figure 5.10 shows a scenario with HIAS. We assume that a placement components group contains two components, `A`, and `B` that will be replaced by `A'`, and `B'`, respectively, and the component `A` is a frontier component. Transaction T0 is initiated in `A` at t0, and initiates a consequent transaction, T1 in `B`. At t1, the *Reconfigurator* isolates the placement components group, and immediately activates the components, `A'`, `B'`. The transactions initiated at t0 in the group will be aborted. They must be replayed at t2 by the replacing components in order to ensure system consistency.

Soft isolation adaptation strategy (SIAS): In this strategy, component barrier applies a set of filtering rules to ensure that running transactions complete before applying changes in the architecture. Two types of functional messages pass the barriers:

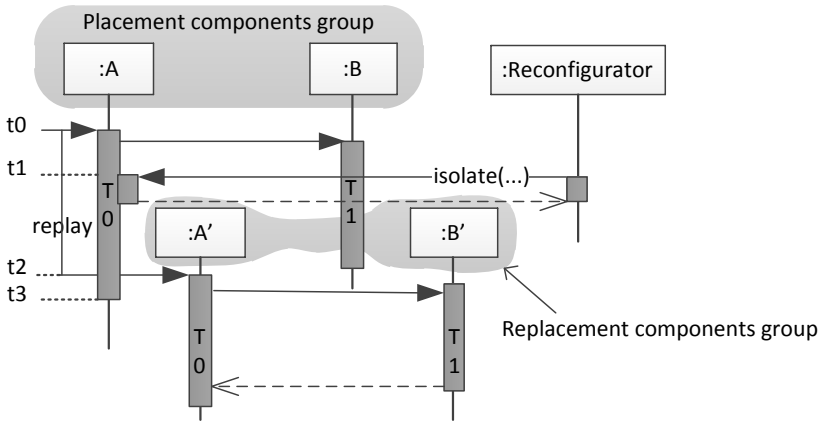


Figure 5.10 – An adaptation with HIAS

1. Messages from components in the placement components group
2. External messages already processed by a starter component but not managed by the ender component.
3. All other messages are blocked at frontier components of the group.

In this strategy, changes on the architecture have to wait till all running transactions complete. Placement components group is not completely isolated from the rest of the system and no store-and-replay mechanism is needed. However, the strategy requires a *Transaction Manager* component to manage transactional dependencies.

Figure 5.11 shows an adaptation scenario similar to the one in Figure 5.10, but with SIAS. The *Reconfigurator* isolates the placement components group and wait till when all components are *free* (t3 in the figure). At this moment, components replacement is realized. Compared to HIAS, the moment to replace components is later than the one in HIAS (t2 in the figure).

	HIAS	**SIAS**
Development time	+	++
Reactivity	++	+

Table 5.2 – A comparison between HIAS and SIAS

Table 5.2 summarizes the characteristics of both strategies. To choose one of them when designing an adaptive software, one may consider two criteria: development time and reactivity on adaptation. HIAS is more expensive when

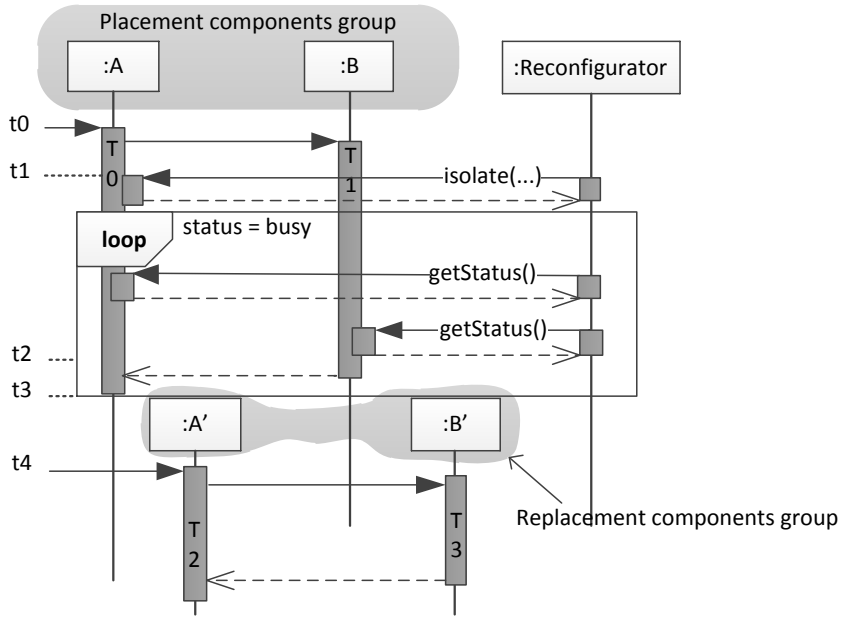


Figure 5.11 – An adaptation with SIAS

considering development cost as store-and-replay mechanism may be complex. For SIAS the *Transaction Manager* component and barriers are integrated in our approach. On the other hand, HIAS is more reactive than SIAS to execute changes. Therefore, if the probability of having long ongoing transactions in a placement components group is important or reactivity on adaptation is critical, developing a HIAS may be a better approach. On the contrary, if reactivity is not an issue or transactions are short-term or seldom, the SIAS strategy may be used.

Our approach is based on SIAS. The main issue is being able to distinguish which messages have to be blocked by the filtering barrier. The next section aims at identifying the rules to apply and the way to implement them.

5.3.2 Isolation of Placement Components Groups

We have distinguished two types of placement components groups: with or without ***dependsOn*** constraints.

5.3.2.1 Placement Components Groups without *dependsOn* Constraints

The isolation of a placement components group without ***dependsOn*** constraints is shown in Figure 5.12. Once an adaptation process has started, the *Reconfigurator* asks the group to isolate itself from the system. Frontier components (components `A`, `B` in the figure) activate their barrier thus blocking all messages coming from the outside of the group (messages coming from ①, and ② in the figure). Messages coming from internal components pass the barrier so that ongoing transactions complete. For example, the barrier of component `B` allows messages from `C` to go through it.

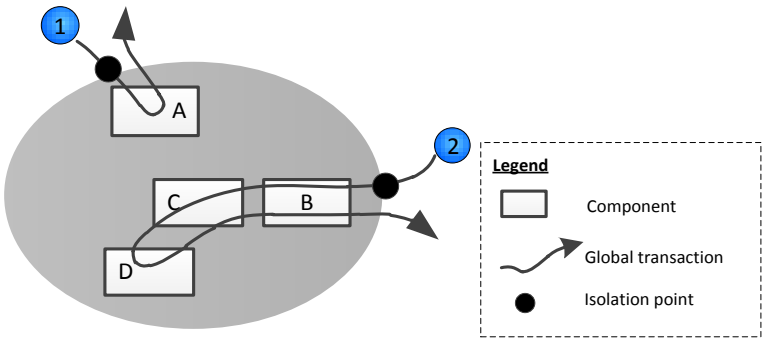


Figure 5.12 – Isolation of a placement components group without ***dependsOn*** constraints

In order to distinguish internal and external messages in frontier components, the *Reconfigurator* should provide the set of components in the group to all frontier components. External messages blocked in the barrier will be transfered to new replacing components that will process them.

5.3.2.2 Placement Components Groups with *dependsOn* Constraints

The isolation of a placement components group with ***dependsOn*** constraints is shown in Figure 5.13. In the group, transactions in `F`, and `I` are dependent on transactions in `H`, and `J`, respectively.

As for groups without ***dependsOn*** constraints, the barrier of all frontier components is activated to isolate the group and internal messages should pass the barrier. However, external messages that have been processed by a starter component should also pass the barrier to complete ongoing transactions in the dependent components (messages from ④, and ⑥ in Figure 5.13). Other

external messages should be blocked (messages from ③, and ⑤ in Figure 5.13).

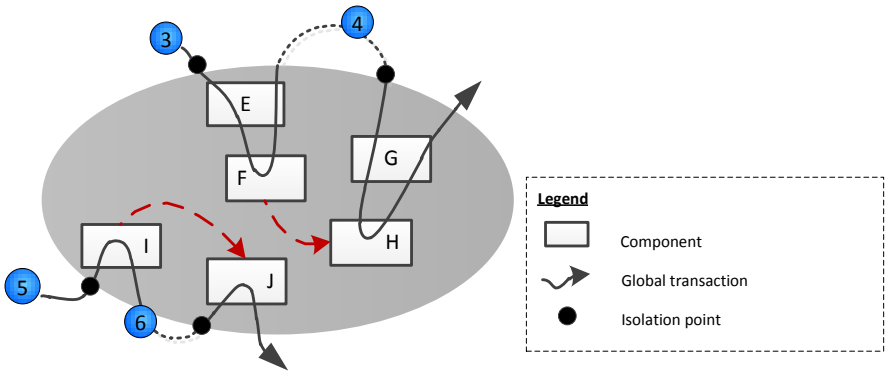


Figure 5.13 – Isolation of a placement components group with ***dependsOn*** constraints

As for placement components groups without ***dependsOn*** constraints, the *Reconfigurator* component sends the list of components in the group to frontier components. To distinguish external messages to be blocked or not, each frontier component relies on the header of each message. As mentioned in Section 5.2.3.1, a starter component (`F`, and `I`) is able to generate the global transaction control information, add it into the header of communication messages, and inform the *Transaction Manager* about the beginning of a global transaction. If the header of a message contains global transaction control information, a frontier component should allow the message to pass the barrier. For example, messages from ④, and ⑥ pass the barrier of components `G` and `J`, respectively, as these messages have been processed by a starter component (`F` and `I` in the figure) but not from the ender.

Based on the above analysis, the set of rules for filtering messages at barrier of components is defined as follows:

- ***Rule 1:*** Messages coming from components in the group and those whose header contains global transaction information should pass barriers.
- ***Rule 2:*** All other messages should be blocked.

As the ender component of a transactional dependency removes global transaction information from messages, our filtering rules allow ongoing transactions to finish consistently.

5.3.3 Summary

In this section, we have presented two adaptation strategies, HIAS and SIAS, based on the isolation of placement components groups. A comparison has also been presented to show the advantages and the disadvantages of each strategy. We base our work on the SIAS approach to deeper understand the idea of transactional dependencies and its impact in the component model.

5.4 Adaptation Process

The purpose of this section is to present the whole picture of an adaptation process and to give hints on how a reconfiguration plan can be generated based on the models specified at design time. An overview of the adaptation process was presented in Section 4.4.3 with four activities: specifying a target resolution model, validating it, planning actions in the architecture, and executing them. The previous chapter presented how to specify the resolution model and validate it. The planning and executing activities will be detailed in this section.

5.4.1 Adaptive Software Architecture

Figure 5.14 shows a generic final adaptive software architecture. Compared to the architecture presented in Figure 4.8, we have added ***dependsOn*** constraints in the variability model (the dotted arrows), connections between the *Reconfigurator* and the adaptive product, and the *Transaction Manager* component. *Reconfigurator* and *Transaction Manager* components are connected to the control part of adaptive product components, the first one to control operation in the architecture, the second one to be aware of ongoing global transactions. In the figure, both are connected to the same components in the adaptive product: for both variability has been specified in the variability model and there is a ***dependsOn*** constraint between them.

The *Reconfigurator* component directly connects to the *Transaction Manager* to get information about the set of ongoing transactions in dependent components. It uses this information to determine when starting components replacement in the architecture.

Finally, the *Planner* component is in charge of planning adaptation. When receiving a new resolution model from external agents, it exploits the models in the repository to compute a reconfiguration plan. The process to generate the reconfiguration plan is presented in the next section.

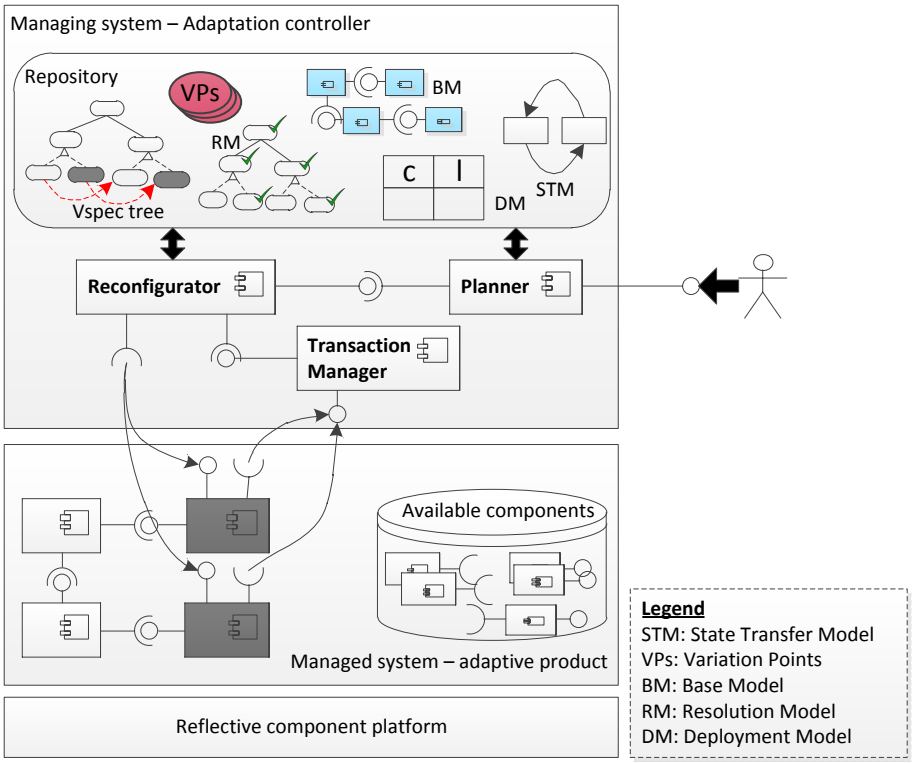


Figure 5.14 – Final general adaptive software architecture

5.4.2 Reconfiguration Plan

The generic form of a reconfiguration plan is presented in Figure 5.15. As shown in the figure, actions in the reconfiguration plan are separated into eight steps:

1. *Isolating the placement components group.* The *Reconfigurator* should be able to identify frontier components in the group to ask for barrier activation. Depending on the type of the barrier (filtering or not), communication messages in the group can be blocked.
2. *Passivating components in the placement components group.* Again, the *Reconfigurator* component needs to know what and where are the components in the group.
3. *Identifying components status information.* To this end, the *Reconfigurator* should be able to know components in the group and to distinguish starter components as their status is managed by the *Transaction Manager* component and not the component itself.
4. *Activating the replacement components group.* The set of replacement

components should be known by the *Reconfigurator* component to activate them. The barrier of components activated in this step is activated to guarantee the order of messages transfered from replaced components and new messages arriving after redirecting connections by the next step.

5. *Connecting the replacement components group.* The *Reconfigurator* component should know connections of the placement components group and those of the replacement components group to redirect them.
6. *Transferring state from the replaced components to the replacing ones.* To this end, the *Reconfigurator* component should be able to map the state from source to destination components.
7. *Reintergrating the replacement components group.* The *Reconfigurator* component should know components in the replacement components group to deactivate their barrier.
8. *Deactivating the placement components group.* The *Reconfigurator* component should know components in the placement components group to deactivate them.

Therefore, depending on the reconfiguration action, the *Planner* component should be able to identify:

- Placement components group,
- Frontier components,
- Starter components,
- Replacement components group, and
- State mapping between replaced and replacing components

Next sections present how this information may be got from the models by the *Planner*. Without this information, it will not be able to generate a plan.

5.4.2.1 Identifying Placement and Replacement Components Group

For an adaptation to be started, a new resolution model needs to be specified. Once the model is provided, the *Planner* combines it with the models in the repository to generate the reconfiguration plan. The first activity in the plan generation process concerns the comparison of two resolution models to identify the placement and replacement components groups.

Figure 5.16 shows an example in which the differences of two architecture configurations can be calculated by comparing two resolution models (RM).

```
<reconfigurationPlan>
  <isolation>
    <!-- frontier components with the placement components group -->
  </isolation>
  <passivation>
    <!-- components in the placement components group-->
  </passivation>
  <getStatus>
    <!-- components in the placement components group for getting status of these
         components and starter components for asking Transaction Manager-->
  </getStatus>
  <activation>
    <!-- replacing components in the replacement components group-->
  </activation>
  <connection>
    <!-- redirection of connections to the replacing components-->
  </connection>
  <stateTransfer>
    <!-- state mapping from the replaced components to the replacing ones-->
  </statetransfer>
  <reintegration>
    <!-- components in the replacement components group-->
  </reintegration>
  <deactivation>
    <!-- components in the placement components group-->
  </deactivation>
</reconfigurationPlan>
```

Figure 5.15 – Generic structure of a generated reconfiguration plan

The differences are propagated to the *VSpec* tree, variation points, and then the base model.

For the first step ①, each resolution model (RM) is searched by using the depth-first search algorithm [2], and represented as an array of TRUE and FALSE values. Resolution models (RM1 and RM2 in the figure) are compared by using a XOR operation. TRUE results indicate a difference between RMs (dotted rectangles in the figure): the *decision* attribute in the *VSpecResolution* has changed from TRUE to FALSE, or from FALSE to TRUE, e.g., the *decision* attribute in the `RLECompression` *VSpecResolution* has been changed from TRUE to FALSE. *VSpecs* that are affected by such a change are deduced in Step ②. According to the changes in *VSpecResolutions*, the selection of *VSpecs* must also be changed: If the value of the *decision* attribute of a *VSpecResolution* changes

2. http://www.cs.yale.edu/homes/aspnes/pinewiki/DepthFirstSearch.html

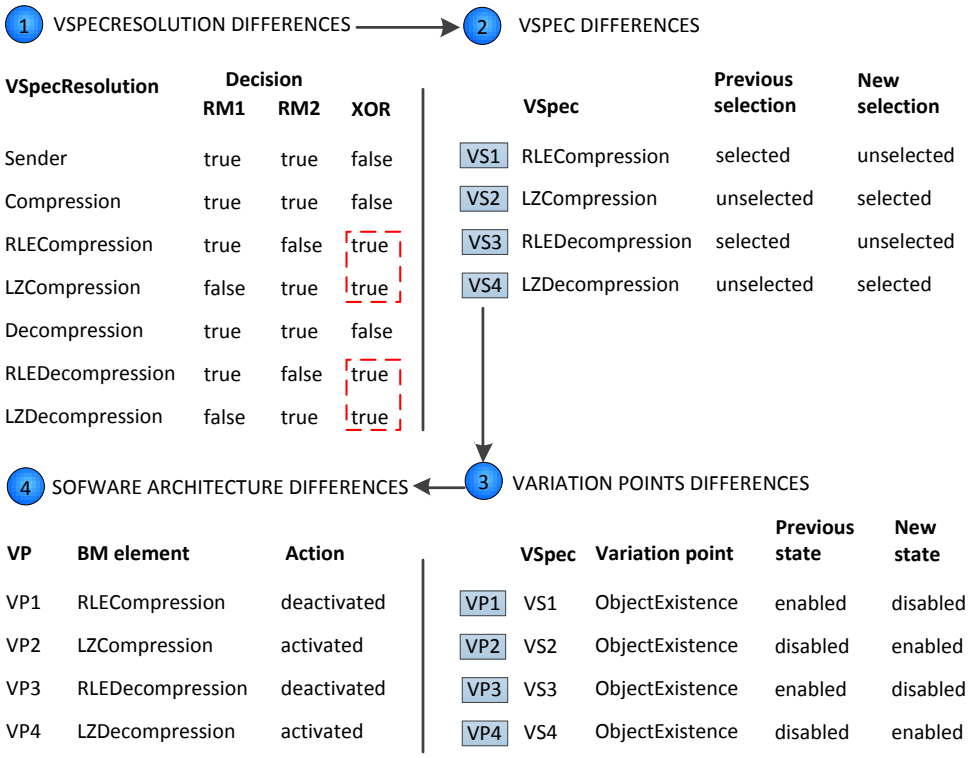


Figure 5.16 – Calculating the differences between two architectural configurations

from TRUE to FALSE, the corresponding *VSpec* previously selected, will be unselected and vice versa (e.g., VS1 changes from selected to unselected in Figure 5.16). Then, variation points bound to such *VSpecs* should be determined in Step ③. If a *VSpec* changes from selected to unselected, the corresponding variation point's state is disabled, and vice versa. For example, VP1 refers to VS1 that has been unselected. Therefore, VP1 will be disabled. In the last step ④, variation points differences are propagated to the base model. Actions to change the architecture can then be determined. In our example, VP1 refers to the `RLECompression` component. As the new state of VP1 is disabled, the `RLECompression` component will be deactivated in the system.

From the results of calculating the differences between architectural configurations, the placement and the replacement components groups can be identified. In our example, the placement components group contains `RLECompression` and `RLEDecompression`, and it will be deactivated. The replacement components group contains `LZCompression` and `LZDecompresssion`, and will be activated in the adaptive product.

Once placement and replacement components groups have been identified, identifying starter, frontier components and connections to be redirected are simple task. For the former, the ***dependsOn*** constraints in the *VSpec* tree are used. For the latter, the base model will allow the *Planner* to identify redirections.

5.4.2.2 State Mapping between Components

An important issue that must be addressed in adaptation to ensure the consistency is the state transfer of components that are replaced. The state includes the local state of all components, i.e., the component attributes, and all messages in transit [Ma 11]. The state can be transfered to new components with or without changes. We propose here a partial solution that is used to validate our approach. Developing this aspect is a perspective of this thesis.

Figure 5.17 shows three cases of state transfer. The upper part shows a case where the state is transferred to the new component as it is. The two lower parts present the cases where the state is transformed via functions $f1$, $f2$ or $f3$.

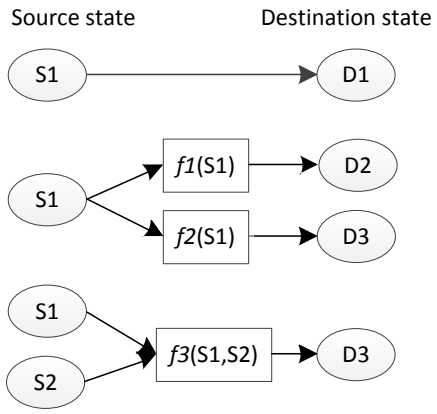


Figure 5.17 – Generality of state transfer

In order to map the state between sources and destinations, we propose a state transfer meta model (see Figure 5.18). The state transfer model is considered as a set of state transfer points. Each point is specified by a destination variable, one or many source variables, and state transfer function. State transfer functions are used to transform the state from sources to destinations. Each of them is described by an operation and its parameters. The parameters refer

to the source variables or constants.

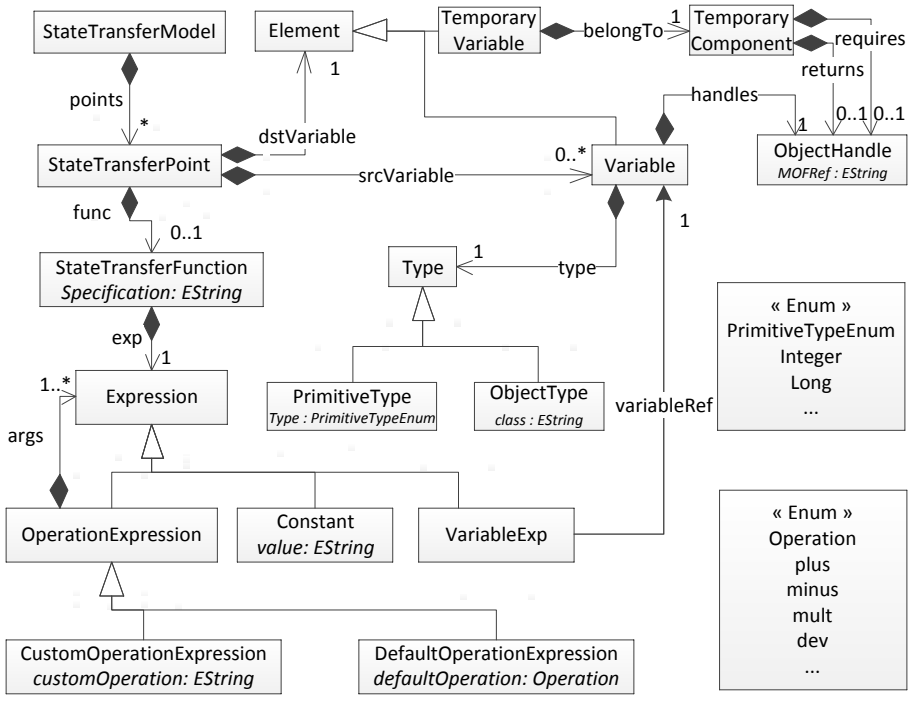


Figure 5.18 – A state transfer meta-model

Several variables in components in the base model may be source variables. The data type of each variable should be identified by using the base model. This allows the *Reconfigurator* to clearly identify the data type of variables.

Destination variables may be existing variables in new components in the base model. If there is no existing variable in the destination component to store the source variables, the state can be lost. If the adaptation requirements must guarantee the state of the system, it should be stored somewhere else. In this case, a temporary variable in a temporary component could be used. If the state to be managed includes communication messages, they should be treated by the new component. Therefore, the temporary component requires services of the new component in the base model.

Figure 5.19 shows an example in which the communication messages have to be transferred from `Server1` to `Server2` during adaptation. We assume that `Server1` has a buffer to store communication messages, and `Server2` has no buffer. A temporary variable in a temporary component is used to store such messages. Action (2) presents the state transfer from `Server1` to `Temporary`. The temporary component requires the service of `Server2` (action (3)), and

then returns the results to the client (actions (4) and (5)). The temporary component will be destroyed when its buffer is empty. This issue is out of the scope of this thesis but it is described to represent the generality of the state transfer model.

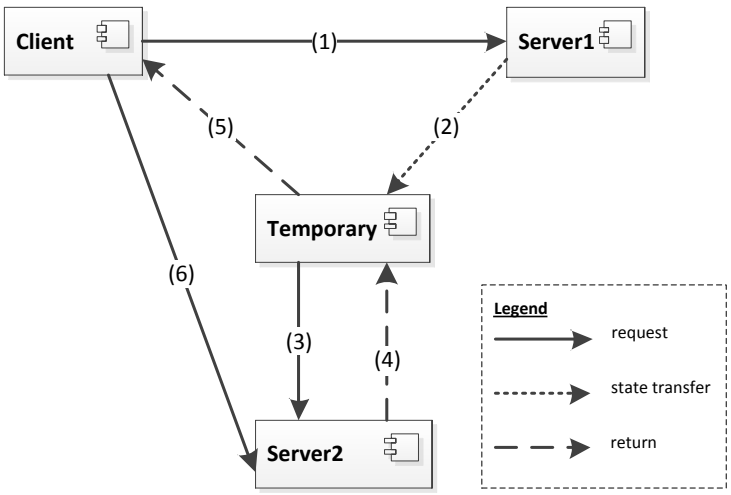


Figure 5.19 – An example with a temporary component when transferring state

The state transfer model for this example is shown in Figure 5.20. It includes a state transfer point. The source variable is mapped to a *buffer* variable in `Server1` and the destination variable is mapped to the *tempBuffer* variable in the `Temporary` component. This component requires services in `Server2` and returns results to `Client`.

In our approach, we assume that components state only includes communication messages. Such messages may be blocked at the barrier of components, and will be transferred between replaced and replacing components during the adaptation process. In terms of implementation, communication messages are stored in a variable in the component implementation. The state transfer model simply describes the mapping of two variables used to store the communication messages between two components. Based on the barrier of components in the placement components group and the state transfer model, state mapping actions can be generated in the reconfiguration plan.

```
<StateTransferModel>
  <points>
    <srcVariable name="buffer">
      <type type="ObjectType" class="Message"/>
      <variableHandle mofRef="Server1"/>
    </srcVariable>
    <dstVariable type="TemporaryVariable" name="tempBuffer">
      <type type="ObjectType" class="Message"/>
      <belongTo name="Temporary">
        <requires mofRef="Server2"/>
        <returns mofRef="Client"/>
      </belongTo>
    </dstVariable>
  </points>
</StateTransferModel>
```

Figure 5.20 – A state transfer model instance

5.4.3 Executing the Reconfiguration Plan

After receiving the reconfiguration plan from the *Planner*, the *Reconfigurator* component executes it. To this end, it has to get the status of components in the placement components group before realizing the component replacement.

In Section 5.4.2, we have presented a reconfiguration plan that includes eight steps. They are executed as follows:

1. *Isolate placement components group.* The *Reconfigurator* takes information about the frontier components described in the reconfiguration plan, then, uses the deployment model to be aware of their location and sends them the isolation command. After receiving the isolation command, the control part in the components will activate the barrier to block messages. As mentioned in Section 5.3.2, messages that satisfy rule 1 will pass the barrier.
2. *Passivate the placement components group.* The *Reconfigurator* uses information in Step 2 of the reconfiguration plan to set components to the passive status. No components in the group will be able to initiate new transactions.
3. *Get status of all components and information about the completion of transactions in the dependent components.* This step has been presented in Figure 5.8. The reconfiguration plan includes the components to be directly accessed by the *Reconfigurator* (to know their status) and those whose status is managed by the *Transaction Manager* (starter compo-

nents). Once all global transactions finish in dependent components and other components of the group are *free*, the component replacement can be realized.

4. *Activate new components.* The replacement components group is included in the plan. The *Reconfigurator* component activates these components. By default, their barrier is activated when activating them.
5. *Redirect connections to the new components.* The *Reconfigurator* redirects the connections to the replacement components group according to Step 5 in the plan.
6. *Transfer state.* All messages stored in the buffer of the frontier components in the placement components group are transfered to the head of buffer in the corresponding components in the replacement components group to guarantee the message order.
7. *Reintegrate components in the replacement components group.* So far, the components in the replacement components group are reintegrated and the barriers are deactivated. The replacement components group is completely joined into the adaptive product.
8. *Deactivate components.* Finally, all components in the placement components group are deactivated.

Reconfiguration actions are realized conforming to the reconfiguration plan. Most these actions are based on the deployment model to locate the components in the adaptive product. In order to realize actions 1, 2, 3, 5, 6, and 7, the *Reconfigurator* interacts with the control part of components. Whereas, the actions 4 and 8 are realized by using the services provided by deployment platforms, e.g., install, start, or stop bundles in OSGi.

5.4.4 Summary

We have presented an adaptation process from an adaptation requirement to realizing reconfiguration actions. An architecture consisting of three main elements, *Reconfigurator, Planner,* and *Transaction Manager* has been presented. The *Planner* generates a reconfiguration plan and exploits models to identify the components involved on each step of the plan. The *Reconfigurator* executes the plan and uses the deployment model to effectively locate components in the system. It connects to the *Transaction Manager* to be aware of the completion of transactions in the dependent components.

Our reconfiguration plan includes state transfer action to guarantee system consistency. Generation of such actions is based on a state transfer model.

Our approach based on models, *Planner*, *Reconfigurator*, *Transaction Manager* components and a component model, is our answer to *Question 4* "How to determine when to replace components and transfer state for consistent dynamic adaptation?".

5.5 Evaluation and Discussion

5.5.1 Evaluation

In order to realize a Proof of Concept (PoC), the case study used in this chapter has been implemented. It is simple but meets the main requirements to illustrate our approach: a placement components group with transactional dependency.

Figure 5.4 shows the variability and the base models of the system. In order to configure the variability model, a resolution model must be specified. We assume that the two components, `LZCompression` and `LZDecompression`, are available at runtime, but are not selected for the product (all other elements are selected in the product).

The AdapSwAG tool described in Section 4.5.1 is reused to generate the product model. As previously mentioned, this tool consists of three modules to validate resolution models, generate product models from the models specified in the domain engineering, and executable code from a product model. In this chapter, the code generation module is extended so that generated code conforms to the component model proposed in Section 5.2.3.3.

Transactional dependencies in the system are specified in Table 5.1. Based on such dependencies, the starter and ender components can be identified: the `RLECompression` and `RLEDecompression` components, respectively. Both need to be connected to the *Transaction Manager* to provide the global transaction control information for managing transactions. This connection is manually completed by product developers. At runtime, the *Transaction Manager* component receives information about the beginning of global transactions from the `RLECompression` component, and their end from the `RLEDecompression` component.

We have implemented our example on top of the iPOJO/OSGi component model [Escoffier 07, The OSGi Alliance 14]. This model allows to build dynamically extensible Java-based applications and facilitates their management by offering features like dependency, component reconfiguration, component factory, and introspection. In order to connect components, the Apache CXF framework [Apache 09] has been used.

```
<reconfigurationPlan:Plan>
  <isolation>
    <actions component="RLECompression" placementComponentGroup="
        RLECompression, RLEDecompression"/>
    <actions component="RLEDecompression" placementComponentGroup="
        RLECompression, RLEDecompression"/>
  </isolation>
  <passivation>
     <actions component="RLECompression"/>
     <actions component="RLEDecompression"/>
  </passivation>
  <getStatus>
    <actions component="RLECompression"/>
    <actions component="RLEDecompression"/>
    <actionsForManager component="RLECompression"/>
  </getStatus>
  <activation>
    <actions component="LZCompression"/>
    <actions component="LZDecompression"/>
  </activation>
  <connection>
    <actions srcComponent="Compression" oldDstComponent="RLECompresison"
        newDstComponent="LZCompression"/>
    <actions srcComponent="Decompression" oldDstComponent="RLEDecompresison"
        newDstComponent="LZDecompression"/>
  </connection>
  <transfer>
    <actions><point>
        <srcVariable name="buffer">
          <variableHandle mofRef="RLECompression"/>
        </srcVariable>
        <dstVariable name="buffer">
          <variableHandle mofRef="LZCompression"/>
        </dstVariable>
    </point></actions>
  </transfer>
  <reintegration>
    <actions component="LZCompression"/>
    <actions component="LZDecompression"/>
  </reintegration>
  <deactivation>
    <actions component="RLECompression"/>
    <actions component="RLEDecompression"/>
  </deactivation>
</reconfigurationPlan:Plan>
```

Figure 5.21 – A generated reconfiguration plan

In our example, a iPOJO component is organized as a package in a Java project. Therefore, there are nine packages corresponding to nine components (`Sender`, `Receiver`, `Compression`, `RLECompression`, `LZCompression`, `Decompression`, `RLEDecompression`, `LZDecompression`, and `Packer`) in the managed system. Each package contains a class for functional behavior and a sub-package for control services. The former is implemented with a barrier, a functional connection, and calculation parts. There are seven different classes used in its implementation. The latter contains four classes: two for control functions, and two for capturing headers of communication messages. In the managing system, there are three components, *Planner*, *Reconfigurator*, and *Transaction Manager*. The *Planner* is implemented with three classes, the *Reconfigurator* with three classes, and the *Transaction Manager* with two classes. Moreover, there are five model instances in the repository supporting the adaptation and a target resolution model instance.

The scenario of adaptation process is described in Figure 5.22. When receiving an adaptation request with a new resolution model from an external agent, e.g., a reconfiguration engineer, the *Planner* generates a reconfiguration plan. The generated reconfiguration plan is shown in Figure 5.21. It includes the eight steps as presented in Section 5.4.2.

After generating the reconfiguration plan, the *Planner* sends it to the *Reconfigurator* to realize the adaptation. For the sake of clarity, the components `Packer`, `Compression`, and `Decompression` are not shown in Figure 5.22. As soon as receiving the reconfiguration plan, the *Reconfigurator* realizes reconfiguration actions according to the eight steps mentioned in Section 5.4.3. As previously, in the step 4 and 8, the *Reconfigurator* uses services provided by the OSGi platform. Thus, its interactions are hidden in the figure.

5.5.2 Discussion

This chapter has presented an approach based on specifying transactional dependencies to realize consistent dynamic adaptation. We have proposed a new constraint in the variability model, ***dependsOn***, to specify transactional dependencies in the variability model. This dependency is exploited to support the management of transactions at runtime. We also propose a component model that contains the necessary control functions to support consistent dynamic adaptation.

In addition, two adaptation strategies have been proposed, HIAS and SIAS. Both are based on the isolation of the placement components group. The former is based on completely isolating the group with a barrier. The latter partially

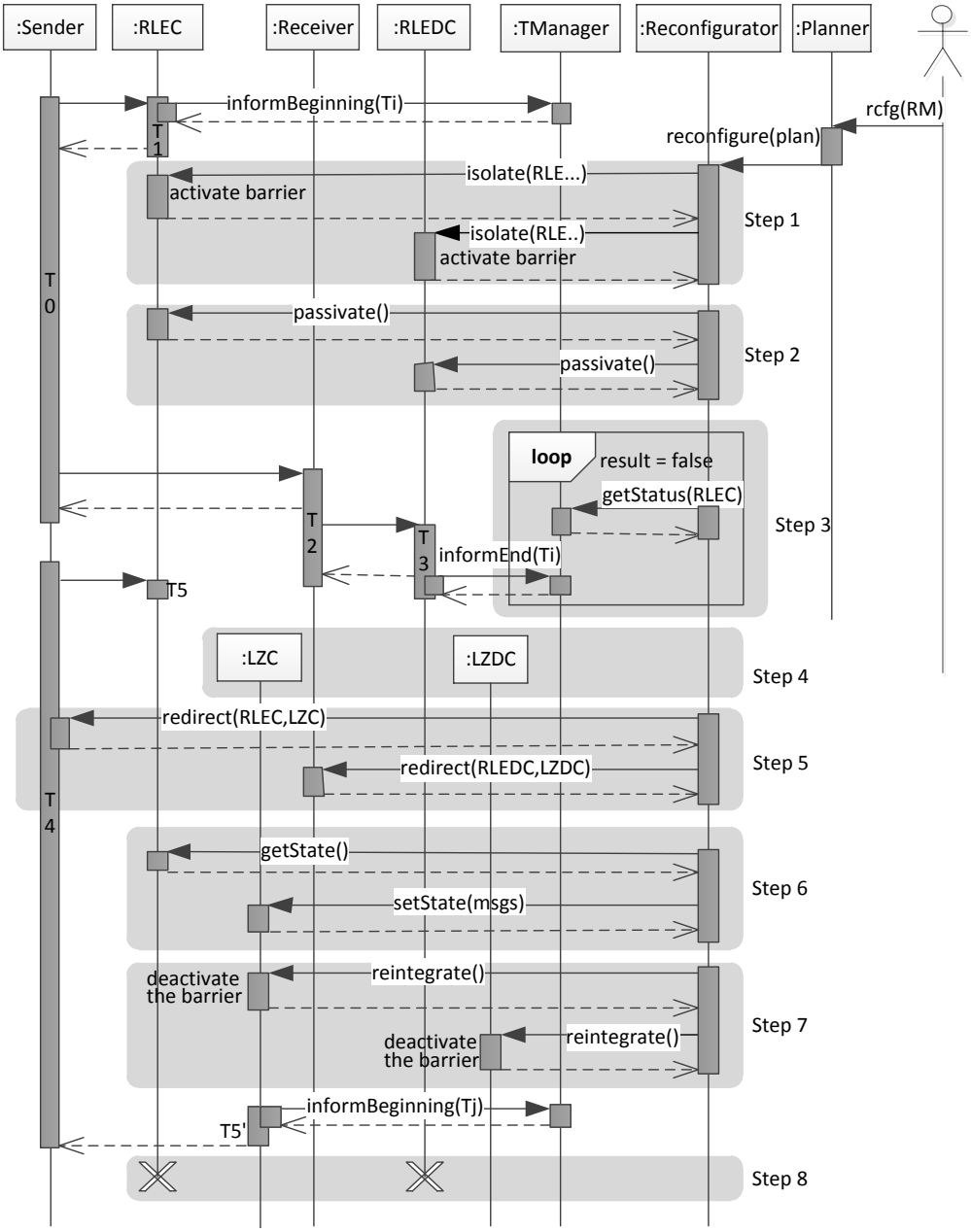


Figure 5.22 – An extract of dynamic adaptation process

isolates the group by using a filtering barrier. A comparison has been made in Table 5.2. For HIAS, the component replacement can be immediately realized after isolation, but the development of mechanisms to ensure consistency is complex. For SIAS, the transaction management allows to ensure consistency during adaptation but component replacement can not start immediately after isolation.

In order to verify the feasibility of our approach, we have extended the AdapSwAG tool. This extension allows the executable code to conform to the proposed component model. Moreover, we have implemented the case study that meets necessary requirements for illustrating the issues of the chapter as mentioned in 5.1.

Part III

Epilogue

Chapter 6

Conclusion and Perspectives

Contents

6.1 Contribution Summary

This thesis has proposed an approach to develop adaptive software architectures. The approach we propose presents a development process from design to runtime. On the one hand, at design time, our approach relies on *specifying the variability* of software product lines and *configuring it* to generate adaptive products. On the other hand, at runtime, a proposed *adaptation mechanism* relies on exploiting the information specified at design time to ensure consistent dynamic adaptation. Particularly, the information about *specifying the transactional dependencies* plays a critical role. Moreover, a *component model* has been proposed to cope with transactional dependencies.

Based on the proposed adaptive software architecture development process, we argue that our approach overcomes the five challenges identified in the Chapter 1 for development of adaptive software architectures: 1) modeling variability and commonality for adaptation, 2) configuring and automatically building adaptive architectures, 3) supporting state transfer, 4) automatically planning adaptation, 5) ensuring consistent dynamic adaptation.

Specifying and Configuring Variability

We have presented a variability-driven development process for building adaptive software architectures. The process is based on CVL tools and meta-models to model the variability and the architecture of an SPL. It defines a set of ordered tasks performed by engineers from the variability specification up to the software architecture. These models are used to configure adaptive products. In order to avoid unnecessary elements to be embedded in the product, the CVL resolution meta-model is extended with a new attribute, ***availabilityAtRuntime***. We have proposed it as an extension of CVL but it can be applied to any model to deal with building the adaptive products.

Developping a Tool Supporting for Validating Configurations of SPL and Generating Adaptive Products

In order to support engineers on building adaptive software architectures, we have developed the AdapSwAG tool to generate the adaptive software architectures. This tool considers the extended CVL model as its input. However, it can be extended to adapt to different inputs such as feature models. In addition, it integrates a module to validate the applicability of a resolution model for configuring a variability model.

Specifying Transactional Dependencies

Identifying the status of the placement components group plays a critical role for ensuring consistent dynamic adaptation. In our approach, this status is identified based on the transaction management. To develop a product supporting transaction management, transactional dependencies should be specified at design time to offer necessary information for designing components of an adaptive product. Therefore, we have proposed ***dependsOn*** constraints. They indicate dependencies between components (called starter and ender components) in terms of transaction: if a transaction is engaged in a starter component, its result must be treated by the ender component before replacing components.

The ***dependsOn*** constraint is used with CVL. However, it can be adapted to other variability models to represent the transactional dependency for building adaptive software architectures.

Component Model for Ensusing Consistent Dynamic Adaptation

In order to support the transaction management, a component model has been proposed with control, connection, calculation, and barrier. The control and connection parts are in charge of observing global transactions and informing the *Transaction Manager* about the beginning of a global transaction in a

starter component and the end of the global transaction in an ender component. Moreover, it needs to provide a status control attribute for managing its status, *free* or *busy*, in all components of the system.

Such component model is exploited to design components when building an adaptive product. At design time, components are assigned different roles (starter or ender) according to the identified transactional dependencies. In so doing, the number of components that has to manage transactions in a placement components group is limited to the starter and ender components.

Furthermore, in order to isolate a placement components group, the component model needs to provide functions that allows to isolate itself from the rest of the system. It is the role of barrier in the component model. When realizing the adaptation process, the barrier of frontier components in the placement components group must be activated to isolate the group.

Adaptation Mechanism for Consistent Dynamic Adaptation

For this purpose, transactional dependencies are exploited at runtime to find when components can be replaced. An adaptation mechanism has been proposed that consists of the necessary activities to replace a placement components group by another. Thanks to the transaction management and the status control attribute, the adaptation mechanism allows to identify the moment when all global transactions finish in the placement components group. This is also the best moment to realize the adaptation actions. Therefore, such an adaptation mechanism ensures the correctness of ongoing activities. Moreover, the state transfer actions allow to guarantee the status of the system.

6.2 Limitations

Our approach for building adaptive software architectures has several limitations. Firstly, the selection of available elements in the adaptive product completely depends on the knowledge of adaptive product designers. Secondly, it does not support the evolution of the product line. Thirdly, it lacks a mechanism to observe informing the *Transaction Manager* about the end of transactions. Finally, there is not the rollback mechanism when arising errors during the adaptation process.

Selection of Available Elements

An adaptive software architecture is generated based on configuring the

variability model. The selection of *VSpecs* in the variability model is decided by the resolution model via the attributes, *decision* and ***availabilityAtRuntime***. However, the availability at runtime of necessary elements depends completely on the knowledge of engineers collected from the requirement process. This approach does not offer criteria to help the engineers on distinguishing the useful and useless elements.

No Support for Evolution

Architectural elements generated in the adaptive product can not change over time, i.e., such elements are foreseen at design time for adaptation. However, a product line may evolve to meet new requirements over many years, and new elements could be arisen in the future and added into the adaptive product. A new element added into the existing product can positively or negatively impact the existing elements. Our approach lacks a process to cope with this aspect.

Lack of an Observation Mechanism

During execution of the adaptive product, the starter and ender components always inform the *Transaction Manager* about the beginning and the end of global transactions. Based on this information, the *Reconfigurator* can identify when they can be replaced. If the starter component has treated a message, but the result message does not arrive at the ender component or the end of global transaction can not arrive at the *Transaction Manager*, the adaptation process will not finish.

Lack of a Rollback Mechanism During Adaptation

The adaptation mechanism we propose is based on managing transactions in a placement components group to ensure consistent dynamic adaptation. However, this adaptation mechanism does not take into account the break arisen during adaptation. Indeed, the system can incorrectly function or even being suspended, if there are errors during adaptation.

6.3 Perspectives

Although the work presented in this thesis covers the needs to build adaptive software architectures, there are also some aspects that could be done to improve our approach.

Towards a Process for the Evolution of Product Line

A product line can evolve over time to meet new requirements. This means that the variability model can grow by adding new *VSpecs*. The new *VSpecs* added can positively impact the existing *VSpecs*. Therefore, the variability model must be modified to adapt to the new *VSpecs* and ensure the consistency of the variability model. In addition, the modification of the variability model leads the changes in the architecture. Consequently, the running system must be modified at runtime as well.

As an ongoing work, we are working on the literature of existing approaches such as [Pleuss 12, Nieke 17] that address this issue. Based on such literature, we see that there is no process to track the evolution from the abstraction specification to the deployment. Therefore, a process should be proposed to meet this issue. This is considered as a short-term perspective of our work.

Supporting Reliable Distributed Adaptation

In the context of a distributed system, an adaptation process may consist of many adaptation subprocesses that are performed in distributed sites. These subprocesses should be coordinated to control the adaptation process. Moreover, a rollback mechanism for returning to the system state before adaptation should be available to handle errors during adaptation.

These aspects have been addressed by our research group. Works in [Phung-khac 10, Huynh 11] addressed the distributed adaptation process and reliable adaptation with a rollback mechanism. However, they do not address the transaction management to determine when changes should be applied.

Extending the State Transfer Aspect

In our approach, a partial solution related to the state transfer have been proposed. However, we are only interested in the state transfer of communication messages after activating components. The state of a component before or during its activation has not been taken into account. In addition, our partial solution addresses changes of state from sources to targets by using state transfer functions. An extended research about this aspect and the proof of its applicability should be studied in the future.

Comparing performance between two strategies, HIAS and SIAS

We have proposed two strategies to realize adaptation. However, our ap-

proach is only focused on SIAS. An extension research about HIAS should be realized. Consequently, a quantitative analysis between HIAS and SIAS is necessary to indicate the effectivity of each strategy according to the development context.

Extending the Prototype and Applying the Approach on a Real System

The prototype implemented in our approach is based on CVL. Thus, it should be improved to adapt to alternative models. Particularly, the AdapSwAG tool should provide functionalities to adapt to different models. Depending on the input models, corresponding functionalities are used.

In addition, we based our work on simple examples to justify our approach. Therefore, a real example should be studied and developed with our approach to overall evaluate it.

Part IV

Bibliography and Appendices

Bibliography

[Allen 97] Robert Allen & David Garlan. *A Formal Basis for Architectural Connection*. ACM Trans. Softw. Eng. Methodol., vol. 6, no. 3, pages 213–249, July 1997. 31

[Apache 09] CXF Apache. *An open source service framework*. See: http://cxf. apache. org, vol. 111, 2009. 125

[Apel 09] Sven Apel & Christiän Kastner. *An Overview of Feature-Oriented Software Development*. Journal of object technoloyg, vol. 8, no. 5, pages 49–84, 8 2009. 20

[Arboleda 13] Hugo Arboleda & Jean-Claude Royer. Model-driven and software product line engineering. John Wiley & Sons, 2013. 17, 19

[Arnold 96] Robert Arnold & Shawn Bohner. Software change impact analysis. Wiley-IEEE Computer Society Press, 1996. 42

[Babar 10] M. A. Babar, L. Chen & F. Shull. *Managing Variability in Software Product Lines*. IEEE Software, vol. 27, no. 3, pages 89–91, 94, May 2010. 17

[Baresi 16] L. Baresi, C. Ghezzi, X. Ma & V. Panzica La Manna. *Efficient Dynamic Updates of Distributed Components through Version Consistency*. IEEE Transactions on Software Engineering, vol. PP, no. 99, pages 1–1, 2016. 60, 63

[Bashari 13] Mahdi Bashari & Ebrahim Bagheri. *Engineering self-adaptive systems and dynamic software product line*. In Proceedings of the 17th International Software Product Line Conference on - SPLC '13, page 285, 2013. 34

[Bass 12] Len Bass, Paul Clements & Rick Kazman. Software architecture in practice. Addison-Wesley Professional, third edition, 2012. 28

[Benavides 05] David Benavides, Pablo Trinidad & Antonio Ruiz-Cortés. *Automated Reasoning on Feature Models*. In 17th International Conference CAiSE 2005, volume 01, pages 491–503. Springer Berlin Heidelberg, 2005. 48

[Bencomo 06] Nelly Bencomo & Gordon S. Blair. *Genie: a Domain-Specific Modeling Tool for the Generation of Adaptive and Reflective Middleware Families*. In 6th OOPSLA Workshop on Domain-Specific Modeling, 2006. 50

[Bencomo 08] Nelly Bencomo. Supporting the modelling and generation of reflective middleware families and applications using dynamic variability. Citeseer, 2008. 11, 34, 50, 54, 58, 63

[Bézivin 01] Jean Bézivin & Olivier Gerbé. *Towards a Precise Definition of the OMG/MDA Framework*. In Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society. 15

[Bialek 04] R. Bialek & E. Jul. *A framework for evolutionary, dynamically updatable, component-based systems*. In Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on, pages 326–331, March 2004. 55, 63

[Bosch 01] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink & Klaus Pohl. *Variability Issues in Software Product Lines*. In 4th International Workshop on Software Product-Family Engineering, pages 13–21, London, UK, 2001. Springer-Verlag. 1, 19, 21

[Brun 09] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè & Mary Shaw. *Software Engineering for Self-Adaptive Systems*. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009. 35

[Buisson 15] Jérémy Buisson, Fabien Dagnat, Elena Leroux & Sébastien Martinez. *Safe reconfiguration of Coqcots and Pycots components*. Journal of Systems and Software, 2015. 33, 59

[Canal 06] Carlos Canal, Juan Manuel Murillo, Pascal Poizat*et al.* *Software Adaptation.* L'objet, vol. 12, no. 1, pages 9–31, 2006. 32

[Capilla 13] Rafael Capilla, Jan Bosch & Kyo-Chul Kang. Systems and software variability management: Concepts, tools and experiences. Springer-Verlag Berlin Heidelberg 2013, Berlin Heidelberg, 2013. 1, 21

[Capilla 14] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés & Mike Hinchey. *An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry.* Journal of Systems and Software, vol. 91, no. 1, pages 3–23, 2014. 48

[Cetina 08a] Carlos Cetina, Joan Fons & Vicente Pelechano. *Applying software product lines to build autonomic pervasive systems.* Proceedings - 12th International Software Product Line Conference, SPLC 2008, no. ii, pages 117–126, 2008. 11, 48, 51, 54, 55, 63

[Cetina 08b] Carlos Cetina, Pablo Trinidad, Vicente Pelechano & Antonio Ruiz-Cortés. *An Architectural Discussion on dynamic Software Product line.* In 2nd SPLC Workshop on Dynamic Software Product Line (DSPL), page 59–68, Limerick, Ireland, Sep 2008. Irish Software Engineering Research Centre (Lero), Irish Software Engineering Research Centre (Lero). 32

[Cetina 09a] Carlos Cetina, Pau Giner, Joan Fons & Vicente Pelechano. *Autonomic Computing through Reuse of Variability Models at Run-Time: The Case of Smart Homes.* Computer, IEEE Computer Society, vol. 42, no. 10, pages 37 – 43, 10 2009. 5, 54, 58

[Cetina 09b] Carlos Cetina, Ø ystein Haugen, Xiaorui Zhang, Franck Fleurey & Vicente Pelechano. *Strategies for variability transformation at run-time.* Proceedings of the 13th International Software Product Line Conference, pages 61–70, 2009. 6, 51, 52

[Cetina 13] Carlos Cetina, Pau Giner, Joan Fons & Vicente Pelechano. *Prototyping Dynamic Software Product Lines to evaluate run-time reconfigurations.* Science of Computer Programming, vol. 78, no. 12, pages 2399 – 2413, 2013. Special Section on International Software Product Line Conference 2010 and Fundamentals

of Software Engineering (selected papers of {FSEN} 2011). 11, 48, 51, 54, 58, 63

[Chapin 01] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil & Wui-Gee Tan. *Types of Software Evolution and Software Maintenance.* Journal of Software Maintenance, vol. 13, no. 1, pages 3–30, January 2001. 33

[Chen 02] Xuejun Chen & Martin Simons. *A Component Framework for Dynamic Reconfiguration of Distributed Systems.* In Proceedings of the IFIP/ACM Working Conference on Component Deployment, CD '02, pages 82–96, London, UK, UK, 2002. Springer-Verlag. 40, 56, 57, 60, 63

[Chen 09] Lianping Chen, Muhammad Ali Babar & Nour Ali. *Variability Management in Software Product Lines: A Systematic Review.* In Proceedings of the 13th International Software Product Line Conference, SPLC '09, pages 81–90, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. 20

[Cheng 02a] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Peter Steenkiste & Ningning Hu. *Software Architecture-Based Adaptation for Grid Computing.* In Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, HPDC '02, pages 389–, Washington, DC, USA, 2002. IEEE Computer Society. 31, 37

[Cheng 02b] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, João Pedro Sousa, Bridget Spitnagel & Peter Steenkiste. *Using Architectural Style As a Basis for System Self-repair.* In Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, WICSA 3, pages 45–59, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V. 31

[Cheng 02c] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, João Pedro Sousa, Bridget Spitznagel, Peter Steenkiste & Ningning Hu. *Software Architecture-Based Adaptation for Pervasive Systems.* In Proceedings of the International Conference on Architecture of Computing

Systems: Trends in Network and Pervasive Computing, ARCS '02, pages 67–82, London, UK, UK, 2002. Springer-Verlag. 2, 37

[Cheng 09] Betty H. C. Cheng, Rogério De Lemos, Holger Giese, Paola Inverardi & Jeff Magee. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Lecture Notes in Computer Science, vol. 5525 LNCS, pages 1–26, 2009. 2, 32, 33, 35

[Clements 01] Paul Clements & Linda Northrop. Software product lines: Practices and patterns. Addison-Wesley Professional, 2001. 17

[Coupaye 07] Thierry Coupaye & Jean-Bernard Stefani. *Fractal Component-based Software Engineering*. In Proceedings of the 2006 Conference on Object-oriented Technology: ECOOP 2006 Workshop Reader, ECOOP'06, pages 117–129, Berlin, Heidelberg, 2007. Springer-Verlag. 2, 23, 30

[Creff 13] Stephen Creff. *A multidimensionnal variability modeling for an incremental product line evolution*. Theses, Université Rennes 1, December 2013. 15

[Czarnecki 00] Krzysztof Czarnecki & Ulrich W. Eisenecker. Generative programming: Methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 18

[Czarnecki 05a] Krzysztof Czarnecki. *Overview of Generative Software Development*. In Proceedings of the 2004 International Conference on Unconventional Programming Paradigms, UPP'04, pages 326–341, Berlin, Heidelberg, 2005. Springer-Verlag. 15, 19

[Czarnecki 05b] Krzysztof Czarnecki, Simon Helsen & Ulrich Eisenecker. *Staged configuration through specialization and multilevel configuration of feature models*. Software Process: Improvement and Practice, vol. 10, no. 2, pages 143–169, 2005. 21

[Czarnecki 05c] Krzysztof Czarnecki & Chang Hwan Peter Kim. *Cardinality-based feature modeling and constraints: a progress report*. In International Workshop on Software Factories at OOPSLA'05, San Diego, California, USA, 2005. ACM, ACM. 20

[Czarnecki 12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid & Andrzej Wasowski. *Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches.* In Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12, pages 173–182, New York, NY, USA, 2012. ACM. 20, 161

[David 09] Pierre-Charles David, Thomas Ledoux, Marc Léger & Thierry Coupaye. *FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures.* annals of telecommunications - annales des télécommunications, vol. 64, no. 1, pages 45–63, 2009. 59

[de Lemos 13] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong & Jochen Wuttke. Software engineering for self-adaptive systems: A second research roadmap, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 33

[De Palma 01] Noël De Palma, Philippe Laumay, Luc Bellissard*et al.* *Ensuring dynamic reconfiguration consistency.* In 6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop, pages 18–24, 2001. 40

[Dobson 06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt & Franco Zambonelli. *A Survey of Autonomic Communications.* ACM Trans. Auton. Adapt. Syst., vol. 1, no. 2, pages 223–259, December 2006. 35, 36, 161

[Elkhodary 09] Ahmed Elkhodary, Sam Malek & Naeem Esfahani. *On the role of features in analyzing the architecture of self-adaptive software systems*. CEUR Workshop Proceedings, vol. 509, pages 41–50, 2009. 49, 53, 59, 63

[Elkhodary 10] Ahmed Elkhodary, Naeem Esfahani & Sam Malek. *FUSION: A Framework for Engineering Self-tuning Self-adaptive Software Systems*. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM. 20, 49, 53, 59, 63

[Escoffier 07] Clement Escoffier, Richard S. Hall & Philippe Lalanda. *iPOJO: An extensible service-oriented component framework*. In IEEE International Conference on Services Computing, numéro Scc, pages 474–481, 2007. 10, 30, 89, 93, 125

[Escoffier 08] Clement Escoffier. *iPOJO: A flexible service-oriented component model for dynamic systems*. Theses, Université Joseph-Fourier - Grenoble I, December 2008. 93

[Farach 98] M. Farach & M. Thorup. *String Matching in Lempel—Ziv Compressed Strings*. Algorithmica, vol. 20, no. 4, pages 388–404, 1998. 8, 97

[Filho 14] João Bosco Ferreira Filho. *Leveraging model-based product lines for systems engineering*. Theses, Université Rennes 1, December 2014. 15

[Floch 06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav & S Intef Ict. *Using Architecture Models for Runtime Adaptability*. IEEE Computer Society, vol. 23, no. 2, pages 62–70, 2006. 2, 51, 53, 63

[France 07] Robert France & Bernhard Rumpe. *Model-driven Development of Complex Software: A Research Roadmap*. In 2007 Future of Software Engineering, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. 15, 16

[Gámez 11] Nadia Gámez, Lidia Fuentes & Miguel A. Aragüez. Autonomic computing driven by feature models and architecture in famiware, pages 164–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 48, 51, 53, 58, 63

[Gámez 15] N. Gámez, L. Fuentes & J. M. Troya. *Creating Self-Adapting Mobile Systems with Dynamic Software Product Lines*. IEEE Software, vol. 32, no. 2, pages 105–112, Mar 2015. 51, 52, 53, 58, 63

[Garlan 00] David Garlan, Robert T. Monroe & David Wile. *Foundations of Component-based Systems*. chapter Acme: Architectural Description of Component-based Systems, pages 47–67. Cambridge University Press, New York, NY, USA, 2000. 3, 2, 23, 31

[Garlan 04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl & Peter Steenkiste. *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure*. Computer, vol. 37, no. 10, pages 46–54, October 2004. 35, 37, 161

[Geihs 09] Kurt Geihs, P Barone, F Eliassen, Jacqueline Floch, R Fricke, E Gjorven, Svein Hallsteinsen, G Horn, N Paspallis, R Reichle & Erlend Stav. *A comprehensive solution for application-level adaptation*. Software - Practice and Experience, vol. 39, pages 661–699, 2009. 32, 33, 51, 53, 56, 57, 63

[Ghafari 12a] Mohammad Ghafari, Pooyan Jamshidi, Saeed Shahbazi & Hassan Haghighi. *An Architectural Approach to Ensure Globally Consistent Dynamic Reconfiguration of Component-based Systems*. In Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE '12, pages 177–182, New York, NY, USA, 2012. ACM. 2, 6, 7, 43, 61, 63, 162

[Ghafari 12b] Mohammad Ghafari, Pooyan Jamshidi, Saeed Shahbazi & Hassan Haghighi. *Safe Stopping of Running Component-Based Distributed Systems: Challenges and Research Gaps*. In Proceedings of the 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE '12, pages 66–71, Washington, DC, USA, 2012. 38, 44

[Ghafari 15] Mohammad Ghafari, Abbas Heydarnoori & Hassan Haghighi. *A Safe Stopping Protocol to Enable Reliable Reconfiguration for Component-Based Distributed Systems*. In Fundamentals of Software Engineering: 6th

International Conference, FSEN 2015, Tehran, Iran, April 22-24, 2015., pages 100–109. Springer International Publishing, 2015. 11, 7

[Gomaa 07] H. Gomaa & M. Hussein. *Model-Based Software Design and Adaptation*. In International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07), pages 7–7, May 2007. 48, 54, 60, 63

[Greenfield 04] Jack Greenfield. Software factories: Assembling applications with patterns, models, frameworks, and tools, pages 304–304. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. 16

[Grondin 08] Guillaume Grondin, Noury Bouraqadi & Laurent Vercouter. *Component reassembling and state transfer in MaDcAr-based self-adaptive software*. In 46th International Conference, TOOLS EUROPE 2008, numéro 11, pages 258–277, Zurich, Switzerland, 2008. Springer Berlin Heidelberg. 1, 44, 55, 57, 63

[Gurp 01] Jilles Van Gurp, Jan Bosch & Mikael Svahnberg. *On the Notion of Variability in Software Product Lines*. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture, WICSA '01, pages 45–, Washington, DC, USA, 2001. IEEE Computer Society. 17

[Hallsteinsen 06] Svein Hallsteinsen, Erlend Stav, Arnor Solberg & Jacqueline Floch. *Using Product Line Techniques to Build Adaptive Systems*. In Proceedings of the 10th International on Software Product Line Conference, SPLC '06, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society. 34

[Hallsteinsen 08] S. Hallsteinsen, M. Hinchey, Sooyong Park & K. Schmid. *Dynamic Software Product Lines*. Computer, vol. 41, no. 4, pages 93–95, April 2008. 2, 34

[Harsu 02] Maarit Harsu. *A Survey on Domain Engineering*. Technical report, Tampere University of Technology, Institute of Software Systems, 2002. 18

[Haugen 08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen & Andreas Svendsen. *Adding Standardized Variability to Domain Specific Languages*. In Proceedings of the 2008 12th International Software Product Line Conference, SPLC '08, pages 139–148,

Washington, DC, USA, 2008. IEEE Computer Society. 51

[Haugen 13] Oystein Haugen, Andrzej Wkasowski & Krzysztof Czarnecki. *CVL: Common Variability Language*. In Proceedings of the 17th International Software Product Line Conference, SPLC '13, pages 277–277, New York, NY, USA, 2013. 48, 91

[Heimbigner 02] Dennis Heimbigner & Alexander Wolf. *Intrusion management using configurable architecture models*. Technical report, DTIC Document, 2002. 37

[Hinchey 12] Mike Hinchey, Sooyong Park & Klaus Schmid. *Building Dynamic Software Product Lines*. Computer, vol. 45, no. 10, pages 22–26, 2012. 2, 34

[Hussein 11] Mohamed Hussein & Hassan Gomaa. *An architecture-based dynamic adaptation model and framework for adaptive software systems*. In 9th IEEE/ACS International Conference on Computer Systems and Applications, pages 165–172, 2011. 2

[Huynh 11] Ngoc-Tho Huynh, An Phung-Khac & Maria-Teresa Segarra. *Towards Reliable Distributed Reconfiguration*. In Adaptive and Reflective Middleware on Proceedings of the International Workshop, ARM '11, pages 36–41, New York, NY, USA, 2011. ACM. 137

[Huynh 16a] N. T. Huynh, M. T. Segarra & A. Beugnard. *A development process based on variability modeling for building adaptive software architectures*. In 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), volume 8, pages 1715–1718. FedCSIS2016, Sept 2016. 6

[Huynh 16b] N. T. Huynh, M. T. Segarra & A. Beugnard. *Ensuring consistent dynamic adaptation: An approach from design to runtime*. In 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), pages 1–8, Nov 2016. 7

[IBM 05] *An Architectural Blueprint for Autonomic Computing*. Technical report, IBM, June 2005. 36

[Kang 90] Kyo Kang, Sholom Cohen, James Hess, William Novak & A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute,

Carnegie Mellon University, Pittsburgh, PA, 1990. 11, 6, 20, 21, 48, 50, 52

[Kelly 07] Steven Kelly & Juha-Pekka Tolvanen. Domain-specific modeling: Enabling full code generation. John Wiley & Sons, Inc., 2007. 16, 17

[Kephart 03] J. O. Kephart & D. M. Chess. *The vision of autonomic computing*. Computer, vol. 36, no. 1, pages 41–50, Jan 2003. 35, 36, 161

[Ketfi 02] Abdelmadjid Ketfi, Noureddine Belkhatir & Pierre-Yves Cunin. *Automatic Adaptation of Component-based Software: Issues and Experiences*. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 3, PDPTA '02, pages 1365–1371. CSREA Press, 2002. 40

[Kim 04] Minseong Kim & Sooyong Park. *Goal and scenario driven product line development*. In 11th Asia-Pacific Software Engineering Conference, pages 584–585, Nov 2004. 34

[Kleppe 03] Anneke G Kleppe, Jos B Warmer & Wim Bast. Mda explained: the model driven architecture: practice and promise. Addison-Wesley Professional, 2003. 15, 16

[Kramer 90] Jeff Kramer & Jeff Magee. *The Evolving Philosophers Problem: Dynamic Change Management*. IEEE Transaction on Software Engineering, vol. 16, no. 11, pages 1293–1306, 1990. 2, 11, 3, 6, 38, 39, 41, 42, 99, 161

[Křikava 13] Filip Křikava. *Domain-specific modeling language for self-adaptive software system architectures*. Theses, Université Nice Sophia Antipolis, November 2013. 35

[Kruchten 06] Philippe Kruchten, Henk Obbink & Judith Stafford. *The Past, Present, and Future for Software Architecture*. IEEE Softw., vol. 23, no. 2, pages 22–30, March 2006. 28

[Kühne 06] Thomas Kühne. *Matters of (Meta-) Modeling*. Software & Systems Modeling, vol. 5, no. 4, pages 369–385, 2006. 15

[Lee 06] Jaejoon Lee & Kyo C. Kang. *A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering*. In Proceedings of

the 10th International on Software Product Line Conference, SPLC '06, pages 131–140, Washington, DC, USA, 2006. IEEE Computer Society. 11, 49, 59, 63

[Léger 10] Marc Léger, Thomas Ledoux & Thierry Coupaye. *Reliable Dynamic Reconfigurations in a Reflective Component Model*. In Proceedings of the 13th International Conference on Component-Based Software Engineering, CBSE'10, pages 74–92, Berlin, Heidelberg, 2010. Springer-Verlag. 59

[Li 11] Wei Li. *Evaluating the impacts of dynamic reconfiguration on the QoS of running systems*. Journal of Systems and Software, vol. 84, no. 12, pages 2123 – 2138, 2011. 41

[Li 12] W. Li. *QoS Assurance for Dynamic Reconfiguration of Component-Based Software Systems*. IEEE Transactions on Software Engineering, vol. 38, no. 3, pages 658–676, May 2012. 38

[Luckham 95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan & Walter Mann. *Specification and Analysis of System Architecture Using Rapide*. IEEE Trans. Softw. Eng., vol. 21, no. 4, pages 336–355, April 1995. 31

[Ludewig 03] Jochen Ludewig. *Models in software engineering – an introduction*. Software and Systems Modeling, vol. 2, no. 1, pages 5–14, 2003. 15

[Ma 11] Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna & Jian Lu. *Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems*. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 245–255, New York, NY, USA, 2011. ACM. 6, 7, 39, 40, 44, 60, 63, 120, 162

[Magee 96] Jeff Magee & Jeff Kramer. *Dynamic Structure in Software Architectures*. SIGSOFT Softw. Eng. Notes, vol. 21, no. 6, pages 3–14, October 1996. 31

[Martinez 15a] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein & Y. l. Traon. *Automating the Extraction of Model-Based Software Product Lines from Model Variants (T)*. In

2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 396–406, Nov 2015. 76

[Martinez 15b] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein & Yves Le Traon. *Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach*. In Proceedings of the 19th International Conference on Software Product Line, SPLC '15, pages 101–110, New York, NY, USA, 2015. ACM. 76

[Martinez 15c] Sébastien Martinez, Fabien DAGNAT & Jérémy Buisson. *Pymoult : On-Line Updates for Python Programs*. In ICSEA 2015 : 10th International Conference on Software Engineering Advances, pages 80 – 85, Barcelone, Spain, November 2015. 3

[Mauro 16] Jacopo Mauro, Michael Nieke, Christoph Seidl & Ingrid Chieh Yu. *Context Aware Reconfiguration in Software Product Lines*. In Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '16, pages 41–48, New York, NY, USA, 2016. ACM. 50, 53, 63

[Medvidovic 00] Nenad Medvidovic & Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Trans. Softw. Eng., vol. 26, no. 1, pages 70–93, January 2000. 31

[Mizouni 14] Rabeb Mizouni, Mohammad Abu Matar, Zaid Al Mahmoud, Salwa Alzahmi & Aziz Salah. *A framework for context-aware self-adaptive mobile applications {SPL}*. Expert Systems with Applications, vol. 41, no. 16, pages 7549 – 7564, 2014. 49, 54, 63

[Moo-Mena 07] Francisco José Moo-Mena. *Modélisation des architectures logicielles dynamiques : application à la gestion de la qualité de service des applications à base de services web*. Theses, Institut National Polytechnique de Toulouse - INPT, April 2007. 28

[Morin 08] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen & Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability, pages 782–796. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 51, 53, 58, 63

[Morin 09a] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey & Arnor Solberg. *Models At Runtime to support dynamic adaptation*. IEEE Computer Society, vol. 42, no. October, pages 44–51, 2009. 11, 51, 58, 63

[Morin 09b] Brice Morin, Olivier Barais, Gregory Nain & Jean-Marc Jezequel. *Taming Dynamically Adaptive Systems using models and aspects*. 31st International Conference on Software Engineering, pages 122–132, 2009. 38, 50, 53, 58, 63

[Muñoz 05] Javier Muñoz & Vicente Pelechano. *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005. Proceedings*. pages 342–356, 2005. 48, 51

[Nieke 17] Michael Nieke, Gil Engel & Christoph Seidl. *DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines*. In Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS '17, pages 92–99, New York, NY, USA, 2017. ACM. 50, 53, 63, 137

[OMG 03] OMG. *Object Management Group. MDA Guide Version 1.0.1*. Document, http://www.omg.org/cgi-bin/doc?omg/03-06-01, 06 2003. 15, 16

[OMG 12] OMG. *Common Variability Language (CVL)*. OMG Revised Submission, 2012. 2, 3, 6, 22, 23, 24, 26, 51, 78, 81, 161, 162

[Oreizy 98] Peyman Oreizy, Nenad Medvidovic & Richard N. Taylor. *Architecture-based Runtime Software Evolution*. In Proceedings of the 20th International Conference on Software Engineering, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society. 33, 38

[Oreizy 99] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum & A. L. Wolf. *An architecture-based approach to self-adaptive software*. IEEE Intelligent Systems and their Applications, vol. 14, no. 3, pages 54–62, May 1999. 1, 32, 33, 37

[Parlavantzas 05] Nikos Parlavantzas. *Constructing modifiable middleware with component frameworks*. PhD thesis, University of Lancaster, 2005. 34

[Parra 11a] Carlos Parra. *Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations*. Theses, Université des Sciences et Technologie de Lille - Lille I, March 2011. 51, 54, 58, 63

[Parra 11b] Carlos Parra, Xavier Blanc, Anthony Cleve & Laurence Duchien. *Unifying design and runtime software adaptation using aspect models*. Science of Computer Programming, vol. 76, no. 12, pages 1247 – 1260, 2011. Special Issue on Software Evolution, Adaptability and Variability. 51

[Pascual 13] Gustavo G. Pascual, Mónica Pinto & Lidia Fuentes. *Run-Time Support to Manage Architectural Variability Specified with CVL*. In Proceedings of the 7th European Conference on Software Architecture, ECSA'13, pages 282–298, Berlin, Heidelberg, 2013. Springer-Verlag. 51

[Pascual 14] Gustavo Pascual, Mónica Pinto & Lidia Fuentes. *Self-adaptation of mobile systems driven by the Common Variability Language*. Future Generation Computer Systems, 2014. 11, 6, 33, 34, 51, 52, 58, 60, 63

[Perrouin 08] G. Perrouin, J. Klein, N. Guelfi & J. M. Jézéquel. *Reconciling Automation and Flexibility in Product Derivation*. In 2008 12th International Software Product Line Conference, pages 339–348, Sept 2008. 21

[Perry 92] Dewayne E. Perry & Alexander L. Wolf. *Foundations for the Study of Software Architecture*. SIGSOFT Softw. Eng. Notes, vol. 17, no. 4, pages 40–52, October 1992. 27

[Phung-khac 10] An Phung-khac. *A Model-driven Feature-based Approach to Runtime Adaptation of Distributed Software Architectures*. Thesis, 2010. 5, 33, 49, 54, 57, 58, 63, 75, 137

[Pleuss 12] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer & Stefan Kowalewski. *Model-driven support for product line evolution on feature level*. Journal of Systems and Software, vol. 85, no. 10, pages 2261 – 2274, 2012. Automated Software Evolution. 137

[Pohl 05] Klaus Pohl, Günter Böckle & Frank van der Linden. Software product line engineering: Foundations, principles, and techniques. Springer-Verlag Berlin Heidelberg, 2005. 11, 2, 6, 17, 18, 48, 50, 161

[Polakovic 08] Juraj Polakovic & Jean-Bernard Stefani. *Architecting reconfigurable component-based operating systems.* Journal of Systems Architecture, vol. 54, no. 6, pages 562 – 575, 2008. Selection of best papers from the 32nd {EUROMICRO} Conference on 'Software Engineering and Advanced Applications' (SEAA 2006). 56, 60, 63

[Rasche 03] A. Rasche & A. Polze. *Configuration and dynamic reconfiguration of component-based applications with Microsoft .NET.* In Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on, pages 164–171, May 2003. 38

[Rubin 13] Julia Rubin & Marsha Chechik. *N-way Model Merging.* In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 301–311, New York, NY, USA, 2013. ACM. 76

[Salehie 09] Mazeiar Salehie & Ladan Tahvildari. *Self-adaptive Software: Landscape and Research Challenges.* ACM Trans. Auton. Adapt. Syst., vol. 4, no. 2, pages 14:1–14:42, May 2009. 33, 35

[Saller 13] Karsten Saller, Malte Lochau & Ingo Reimund. *Context-aware DSPLs: Model-based Runtime Adaptation for Resource-constrained Systems.* In Proceedings of the 17th International Software Product Line Conference Co-located Workshops, SPLC '13 Workshops, pages 106–113, New York, NY, USA, 2013. ACM. 49, 53, 59, 63

[Saller 15] Karsten Saller. *Model-Based Runtime Adaptation of Resource Constrained Devices.* PhD thesis, Technische Universität, Darmstadt, January 2015. 49

[Schmerl 02] Bradley Schmerl & David Garlan. *Exploiting Architectural Design Knowledge to Support Self-repairing Systems.* In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE '02, pages 241–248, New York, NY, USA, 2002. ACM. 28, 31

[Schmid 04] Klaus Schmid & Isabel John. *A customizable approach to full lifecycle variability management.* Science of Computer Programming, vol. 53, no. 3, pages 259 – 284, 2004. Software Variability Management. 2

[Schmidt 06] Douglas C. Schmidt. *Guest Editor's Introduction: Model-Driven Engineering.* Computer, vol. 39, no. undefined, pages 25–31, 2006. 15, 16, 17

[Seinturier 12] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni & Jean-Bernard Stefani. *A component-based middleware platform for reconfigurable service-oriented architectures.* Software: Practice and Experience, vol. 42, no. 5, pages 559–583, 2012. 56, 60

[Shaw 96a] Mary Shaw. *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status.* In Selected Papers from the Workshop on Studies of Software Design, ICSE '93, pages 17–32, London, UK, UK, 1996. Springer-Verlag. 29

[Shaw 96b] Mary Shaw & David Garlan. Software architecture: Perspectives on an emerging discipline. Prentice Hall, 1996. Prentice Hall Ordering Information. 27

[Sinnema 04] Marco Sinnema, Sybren Deelstra, Jos Nijhuis & Jan Bosch. *COVAMOF: A Framework for Modeling Variability in Software Product Families.* In In Proceedings of the Third International Software Product Line Conference SPLC, 2004. 19

[Stoicescu 12] Miruna Stoicescu, Jean-Charles Fabre & Matthieu Roy. *From Design for Adaptation to Component-Based Resilient Computing.* In Dependable Computing (PRDC), 2012 IEEE 18th Pacific Rim International Symposium on, pages 1–10, Niigata, 2012. IEEE. 56, 57, 60, 63

[Sztipanovits 97] J. Sztipanovits & G. Karsai. *Model-integrated computing.* Computer, vol. 30, no. 4, pages 110–111, Apr 1997. 16

[Szyperski 02] Clemens Szyperski. Component software: Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 28

[Taylor 09] R. N. Taylor, N. Medvidovic & E. M. Dashofy. Software architecture: Foundations, theory, and practice. Wiley Publishing, 2009. 29

[The OSGi Alliance 14] The OSGi Alliance. The OSGi alliance OSGi Core, release 6. OSGi edition, 2014. 125

[Trinidad 07] Pablo Trinidad, Antonio Ruiz Cortés, Joaquín Peña & David Benavides. *Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines*. In 1st SPLC Workshop on Dynamic Software Product Line (DSPL), pages 51–56, Kyoto, Japan, 2007. 5, 48, 63

[Truyen 06] Frank Truyen. *The Fast Guide to Model Driven Architecture, The Basics of Model Driven Architecture*. Cephas Consulting Corp, anuary 2006. 15

[Vandewoude 03] Yves Vandewoude & Yolande Berbers. *Meta-Model Driven Methodology for State Transfer in Component-Oriented Systems*. Technical report, Warshau, Poland, 2003. 44

[Vandewoude 04] Yves Vandewoude & Yolande Berbers. *Supporting Run-Time Evolution In Seescoa*. J. Integr. Des. Process Sci., vol. 8, no. 1, pages 77–89, January 2004. 60

[Vandewoude 05] Yves Vandewoude & Yolande Berbers. *Component state mapping for runtime evolution*. In In Proceedings of the 2005 International Conference on Programming Languages and Compilers, pages 230–236, 2005. 2, 44, 56, 57, 60, 63

[Vandewoude 07] Yves Vandewoude, Peter Ebraert, Yolande Berbers & Theo D'Hondt. *Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates*. IEEE Transaction on Software Engineering, vol. 33, no. 12, pages 856–868, 2007. 11, 3, 6, 38, 42, 60, 63, 99

[Vestal 93] Steve Vestal. *A cursory overview and comparison of four architecture description languages*. Technical report, Citeseer, 1993. 30

[Watson 03] V. Watson. *Run-length encoding*, July 24 2003. US Patent App. 10/143,542. 8, 97

[Weiss 99] David M. Weiss & Chi Tau Robert Lai. Software product-line engineering: A family-based software development process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 17

[Zhang 06] Ji Zhang & Betty H. C. Cheng. *Model-based Development of Dynamically Adaptive Software*. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM. 55, 63

List of Figures

List of Tables

Appendices

Appendix A

Publications

The following papers related to our work have been published and are being prepared:

1. Ngoc-Tho Huynh, An Phung-Khac, and Maria-Teresa Segarra. 2011. *Towards reliable distributed reconfiguration.* In Adaptive and Reflective Middleware on Proceedings of the International Workshop (ARM '11). ACM, New York, NY, USA, 36-41.
2. Ngoc Tho Huynh, Maria Teresa Segarra, Antoine Beugnard. *Reconfiguration d'architecture logicielle : la nature des communications entre composants.* CIEL 2015 : 4ème Conférence en IngénieriE du Logiciel, Jun 2015, Bordeaux, France. 2015.
3. Ngoc Tho Huynh, Maria Teresa Segarra, Antoine Beugnard. *A development process based on variability modeling for building adaptive software architectures.* Federated Conference on Computer Science and Information Systems (FedCSIS), Gdansk, 2016, pp. 1715-1718.
4. Ngoc Tho Huynh, Maria Teresa Segarra, Antoine Beugnard. *Ensuring consistent dynamic adaptation: an approach from design to runtime.* AICCSA 2016: 13th ACS/IEEE International Conference on Computer Systems and Applications, 29 november - 02 december 2016, Agadir, Morocco, 2016.
5. Working on an article, Ensuring consistent dynamic adaptation: an approach based on managing transactional dependencies, will be submitted to ACM Transactions on Autonomous and Adaptive Systems (TAAS), 2017.

Appendix B

Implementation Details

In this appendix, we present some implementations related to our approach. On the one hand, the modules in the AdapSwAG tool are detailed. On the other hand, the services in the control part of the component model presented in Section 5.2.3.3 will be detailed in this appendix. Moreover, the implementations of the *Transaction Manager* and *Reconfigurator* are briefly represented as well.

2.1 AdapSwAG Tool

2.1.1 Resolution Model Validation

We apply the algorithm 1 in context of CVL to validate the resolution model. The following procedure is recursively called to search *VSpecResolutions* in the resolution model. In context of this thesis, we are interested in the *Choice VSpec*. Thus, only *ChoiceResolution* is considered in the resolution model.

```
1  String validateNode(VSpecResolution vspecResolution,
2          ArrayList<BCLConstraint> bclConstraints) {
3     if (vspecResolution instanceof ChoiceResolution) {
4        ChoiceResolution choiceResolution = (ChoiceResolution)
             vspecResolution;
5        Choice resolvedChoice = choiceResolution.getResolvedChoice();
6        if (!choiceResolution.isDecision()) { // check impliedByParent
7           if (resolvedChoice.isIsImpliedByParent()) {
8              EObject parent = choiceResolution.eContainer();
9              if (((ChoiceResolution) parent).isDecision()) {
10                return "Error: Conflict with impliedByParent constraint
                      in "
```

```
                    + choiceResolution.getName();
            }
         }
      } else {
         if (resolvedChoice.getGroupMultiplicity() != null) { // check
                                                 // multiplicity
            int min = resolvedChoice.getGroupMultiplicity().getLower();
            int max = resolvedChoice.getGroupMultiplicity().getUpper();
            int count = 0;
            for (VSpecResolution subVSpecResolution : choiceResolution
                  .getChild()) {
               if (subVSpecResolution instanceof ChoiceResolution) {
                  ChoiceResolution subChoiceResolution =
                      (ChoiceResolution) subVSpecResolution;
                  if (subChoiceResolution.isDecision()) {
                     count++;
                  }
               }
            }
            if ((count < min) || (count > max)) {
               return "Error: Conflict with multiplicity constraint in "
                     + resolvedChoice.getName();
            }
         }

         // check implies/excludes
         for (BCLConstraint bclConstraint : bclConstraints) {
            EList<BCLExpression> expressions = bclConstraint
                  .getExpression();
            for (BCLExpression expression : expressions) {
               if (expression instanceof OperationCallExp) {
                  OperationCallExp operationExpression =
                      (OperationCallExp) expression;
                  EList<BCLExpression> arguments = operationExpression
                        .getArgument();
                  VSpec vspec1 = ((VSpecRef) arguments.get(0))
                        .getVSpec();
                  VSpec vspec2 = ((VSpecRef) arguments.get(1))
                        .getVSpec();
                  if ((operationExpression.getOperation().getName()
                        .equals("logImplies"))
                        && (vspec1.getName().equals(resolvedChoice
                              .getName()))) {
                     for (VSpecResolution vspresolution :
                         resolutionList) {
                        if ((vspresolution.getResolvedVSpec()
                              .getName().equals(vspec2.getName()))
```

```
55                              && (!((ChoiceResolution) vspresolution)
56                                    .isDecision())) {
57                            return "Error: Conflict with implies
                                  constraint at "
58                                    + resolvedChoice.getName();
59                            }
60                         }
61                      }
62                      if ((operationExpression.getOperation().getName()
63                            .equals("logExcludes"))
64                            && (vspec1.getName().equals(resolvedChoice
65                                  .getName()))) {
66                         for (VSpecResolution vspresolution :
                               resolutionList) {
67                            if ((vspresolution.getResolvedVSpec()
68                                  .getName().equals(vspec2.getName()))
69                                  && (((ChoiceResolution) vspresolution)
70                                        .isDecision())) {
71                               return "Error: Conflict with excludes
                                     constraint at "
72                                       + resolvedChoice.getName();
73                            }
74                         }
75                      }
76                   }
77                }
78             }
79             if (choiceResolution.getChild() != null) {
80                for (VSpecResolution subVSpecResolution : choiceResolution
81                      .getChild()) {
82                   return validateNode(subVSpecResolution, bclConstraints);
83                }
84             }
85          }
86       }
87       ...
88       return "valid";
89    }
```

2.1.2 Product Model Generation

This module is implemented to generate a product model that contains active components and connections among them, and available components. The following procedure is implemented to identify the active components.

```
ArrayList<ComponentInstance>
    activeComponents(ArrayList<VSpecResolution> vspecResolutions,
        ArrayList<VariationPoint> variationPoints,
            ArrayList<ComponentInstance> componentsInBaseModel) {
   ArrayList<ComponentInstance> componentInstances = new
       ArrayList<ComponentInstance>();
   for (VSpecResolution vspecResolution : vspecResolutions) {
      if ((vspecResolution instanceof ChoiceResolution)) {
         ChoiceResolution choiceResolution = (ChoiceResolution)
             vspecResolution;
         if (choiceResolution.isDecision()) {
            Choice choice = choiceResolution.getResolvedChoice();
            for (VariationPoint variationpoint : variationPoints) {
               if ((variationpoint instanceof ObjectExistence)) {
                  ObjectExistence objExistence = (ObjectExistence)
                      variationpoint;
                  if (objExistence.getBindingChoice().getName()
                        .equals(choice.getName())) {
                     ObjectHandle objHandle = objExistence
                           .getOptionalObject();
                     String mofRefComponent = objHandle.getMOFRef();

                     ComponentInstance activeComponent =
                         returnComponentByName(
                           mofRefComponent, componentsInBaseModel);
                     componentInstances.add(activeComponent);
                  }
               }
            }
         }
      }
   }
   return componentInstances;
}
```

ACME uses two types of connection, *Binding* and *Attachment*. In order to identify the connections among components, two following procedures are implemented.

```
ArrayList<Attachment> activeAttachments(ArrayList<ComponentInstance>
    activeComponents,
        ArrayList<Attachment> attachments) {

   ArrayList<Attachment> activeAttachmentsList = new
       ArrayList<Attachment>();
   for (Attachment attachment : attachments) {
```

```
6        String component = attachment.getComp();
7        for (ComponentInstance componentInstance : activeComponents) {
8           if (componentInstance.getName().equals(component)) {
9              activeAttachmentsList.add(attachment);
10          }
11       }
12    }
13    return activeAttachmentsList;
14 }
```

```
1
2  ArrayList<Binding> activeBindings(ArrayList<ComponentInstance>
       activeComponents,
3        ArrayList<Binding> bindings) {
4
5     ArrayList<Binding> activeBindingsList = new ArrayList<Binding>();
6     for (Binding binding : bindings) {
7        String srcComponent = binding.getCompSrc();
8        String dstComponent = binding.getCompDest();
9        boolean chkSrcComponentInList = false;
10       boolean chkDstComponentInList = false;
11       for (ComponentInstance componentInstance : activeComponents) {
12          if (componentInstance.getName().equals(srcComponent)) {
13             chkSrcComponentInList = true;
14          }
15          if (componentInstance.getName().equals(dstComponent)) {
16             chkDstComponentInList = true;
17          }
18       }
19       if ((chkSrcComponentInList) && (chkDstComponentInList)) {
20          activeBindingsList.add(binding);
21       }
22    }
23    return activeBindingsList;
24 }
```

Finally, the available components are added into the product model according to value of the ***availabilityAtRuntime*** attribute.

```
1  ArrayList<ComponentInstance>
       availableComponents(ArrayList<VSpecResolution> vspecResolutions,
2        ArrayList<VariationPoint> variationPoints,
             ArrayList<ComponentInstance> componentsInBaseModel) {
3     ArrayList<ComponentInstance> inactiveComponentInstances = new
          ArrayList<ComponentInstance>();
4     for (VSpecResolution vspecResolution : vspecResolutions) {
```

```
5       if ((vspecResolution instanceof ChoiceResolution)) {
6          ChoiceResolution choiceResolution = (ChoiceResolution)
               vspecResolution;
7
8          if ((!choiceResolution.isDecision())
9                && (choiceResolution.isAvailabilityAtRuntime())) {
10            Choice choice = choiceResolution.getResolvedChoice();
11            for (VariationPoint variationpoint : variationPoints) {
12
13               if ((variationpoint instanceof ObjectExistence)) {
14                  ObjectExistence objE = (ObjectExistence)
                        variationpoint;
15
16                  if (objE.getBindingChoice().getName()
17                        .equals(choice.getName())) {
18                     ObjectHandle objHandle = objE
19                           .getOptionalObject();
20                     String mofRefComponent = objHandle.getMOFRef();
21                     ComponentInstance activeComponent =
                           returnComponentByName(
22                           mofRefComponent, componentsInBaseModel);
23                     inactiveComponentInstances.add(activeComponent);
24                  }
25               }
26            }
27         }
28      }
29   }
30   return inactiveComponentInstances;
31 }
```

2.2 Transaction Management and Adaptation Controller

2.2.1 Control Service Provided by the Component Model

In order to control components during adaptation, the control part in the component model should provide services. Such services will be invoked by the *Reconfigurator* during adaptation process. They are represented via an interface as follows.

```
1 public interface ControlService {
2   public void isolate(boolean bool, ArrayList<String> componentList);
```

```
3    public void passivate(boolean bool);
4    public void redirect(String oldAddress, String newAddress);
5    public Status getStatus(); //two status: free and busy
6    public CircularFifoQueue <Message> getState();
7    public void setState(CircularFifoQueue <Message> msg);
8  }
```

The implementation of this interface is realized in a Controller class of each component. For example, the isolation service can be implemented in the Controller class as follows.

```
1  public void isolate(boolean bool, ArrayList<String> componentList) {
2    // activation of barrier
3    for (Factory factory : factories) {
4      if (factory.getName().equals("ComponentName")) {
5        ComponentInstance im = (ComponentInstance)
             factory.getInstances().get(0);
6
7        ComponentInstance ci = (ComponentInstance) im;
8        Properties props = new Properties();
9        props.put("isIsolation", bool); //isIsolation is an attribute
             to control the barrier of the "CompnentName" component
10       props.put("componentsInGroup", componentList);
             //componentsInGroup is an attribute to control the
             filtering of the barrier
11       im.reconfigure(props);
12     }
13   }
14 }
```

The controller part exploits understandable (introspection) and reconfigurable (intercession) services in target platforms to control components.

2.2.2 Transaction Manager

This section represents an implementation of *Transaction Manager* component that provides services to receive the beginning and the end of global transactions from dependent components. The global transaction information contains name of component and transaction identification that are defined in a class *TransactionIdentification*. Moreover, the *Transaction Manager* should provide services to the *Reconfigurator* to be known the termination of transactions in the placement components group. Connections among them are implemented using Remote Method Invocation - RMI. We use RMI because of

distinguishing it from the functional connections (CXF) among components.

An interface of the *Transaction Manager* is represented as follows.

```
1 public interface TransactionManagerService {
2    public void informBeginning(TransactionIdentification transactionID)
         throws RemoteException;
3    public void informEnd(TransactionIdentification transactionID)
         throws RemoteException;
4    public boolean getStatus(ArrayList<String> starterComponents) throws
         RemoteException;
5 }
```

The interface consists of three services declared, *informBeginning*, *informEnd*, and *getStatus*. The interface is implemented by the *TransactionManager* class as follows.

```
1  public class TransactionManager implements TransactionManagerService {
2     /*
3      * This class is used to manage transactions processed by dependent
4      * components
5      */
6     String address;
7     int port;
8     ArrayList<TransactionIdentification> transactionList = new
          ArrayList<TransactionIdentification>();
9
10    public TransactionManager() throws RemoteException {
11       this.transactionList = null;
12       Registry registry; // rmi registry for lookup the remote objects.
13       System.setProperty("java.security.policy",
             "java.security.AllPermission");
14       System.setProperty("java.rmi.server.hostname", this.address);
15       try {
16          // create the registry and bind the name and object.
17          registry = LocateRegistry.createRegistry(this.port);
18          registry.rebind("TransactionManager", this);
19       } catch (RemoteException e) {
20          throw e;
21       }
22    }
23
24    @Override
25    public void informBeginning(TransactionIdentification transactionID)
          throws RemoteException {
26       transactionList.add(transactionID);
27    }
```

```
28
29    @Override
30    public void informEnd(TransactionIdentification transactionID)
          throws RemoteException {
31       transactionList.remove(transactionList.indexOf(transactionID));
32    }
33
34    @Override
35    public boolean getStatus(ArrayList<String> starterComponents) throws
          RemoteException
36    {
37       for (TransactionIdentification ti : transactionList) {
38          String componentName = ti.getComponentName();
39          for (String starterComponent : starterComponents) {
40             if (componentName.equals(starterComponent)) {
41                return false;
42             }
43          }
44       }
45       return true;
46    }
47 }
```

2.2.3 Reconfigurator

The *Reconfigurator* needs to provide a service that is invoked by the *Planner*. The plan generated by the *Planner* is considered as a parameter in the service provided by the *Reconfigurator*. Such service is implemented as follows.

```
1  package service;
2
3  public interface ReconfigurationService {
4     public void reconfigure(String planFile);
5  }
```

The *Reconfigurator* reads the plan and realizes adaptation actions. In order to realize actions such as isolate, get status, etc, on components, the *Reconfigurator* needs to connect to the control part of the component and invoke its services. One of actions is illustrated as follows.

```
1  public void isolate(String component, String address, int port,
       ArrayList<String> componentList) {
2     /*
3   * connect to the isolated component
```

```
 */
   ControlService controlService;
   Registry registry;

   try {
      registry = LocateRegistry.getRegistry(address, port);
      controlService = (ControlService) (registry.lookup(component
            + "Controller"));
      controlService.isolate(true, componentList);
   } catch (RemoteException e) {
      e.printStackTrace();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

The function, *isolate*, contains four parameters in which the first and the last ones, *component* and *componentList*, are deduced from the reconfiguration plan, the rest parameters are found by using the deployment model. Other actions that invoke the services provided by the control part are similarly implemented as well.

On the other hand, to identify the termination of transactions in the dependent components, the *Reconfigurator* invokes the service, *getStatus(components)*, provided by the *Transaction Manager* with the parameters of the address of the *Transaction Manager* and a set of starter components.

```
public boolean getStatusInDependentComponents(String address, int
    port, ArrayList<String> starterComponents) {
   TransactionManagerService transactionService;
   Registry registry;
   boolean result = false;
   try {
      registry = LocateRegistry.getRegistry(address, port);
      transactionService = (TransactionManagerService)
          (registry.lookup("TransactionManager"));
      result = transactionService.getStatus(starterComponents);
   } catch (RemoteException e) {
      e.printStackTrace();
   } catch (Exception e) {
      e.printStackTrace();
   }
   return result;
}
```

A TRUE return indicates that this is the moment when global transactions finish in the dependent components.

Furthermore, to implement the functions for activating and deactivating distributed components in the system, we exploit the remote management services that are supported by deployment frameworks, e.g., Apache Felix Framework. These services are implemented using Java Management Extensions (JMX). Thanks to such services, the *Reconfigurator* can remotely manage the components in hosts. For example, the following code is used to activate a component.

```
1  public void activate(String componentName, String host, String
       felixFrameworkUUID) {
2     System.out.println("start the activation action");
3     MBeanServerConnection server = connecToRemoteHost(host);
4     long bundleid = installBundleOnRemoteHost(server, componentName,
          felixFrameworkUUID);
5     ObjectName mbeanName;
6     try {
7        mbeanName = new ObjectName("osgi.core:type=framework,version=1.7,"
8              +"framework=org.apache.felix.framework,uuid="
9              +felixFrameworkUUID);
10       FrameworkMBean osgiFrameworkProxy = JMX.newMBeanProxy(server,
             mbeanName, FrameworkMBean.class);
11       osgiFrameworkProxy.startBundle(bundleId);
12    } catch (MalformedObjectNameException | IOException e) {
13       e.printStackTrace();
14    }
15    System.out.println("finish the activation action");
16 }
```

Résumé

Les logiciels adaptatifs sont une classe de logiciels qui peuvent modifier leur structure et comportement à l'exécution afin de s'adapter à des nouveaux contextes d'exécution. Le développement de logiciels adaptatifs a été un domaine de recherche très actif les dix dernières années. Plusieurs approches utilisent des techniques issues des lignes des produits afin de développer de tels logiciels. Ils proposent des outils, des frameworks, ou des langages pour construire des architectures logicielles adaptatives, mais ne guident pas les ingénieurs dans leur utilisation. De plus, ils supposent que tous les éléments spécifiés à la conception sont disponibles dans l'architecture pour l'adaptation, même s'ils ne seront jamais utilisés. Ces éléments inutiles peuvent être une cause de soucis lors du déploiement sur une cible dont l'espace mémoire est très contraint par exemple. Par ailleurs, le remplacement de composants à l'exécution reste une tâche complexe, elle doit assurer non seulement la validité de la nouvelle version, mais aussi préserver la terminaison correcte des transactions en cours.

Pour faire face à ces problèmes, cette thèse propose un processus de développement de logiciels adaptatifs où les tâches, les rôles, et les artefacts associés sont explicites. En particulier, le processus vise la spécification d'informations nécessaires pour construire des architectures logicielles adaptatives. Le résultat d'un tel processus est une architecture logicielle adaptative qui contient seulement des éléments utiles pour l'adaptation. De plus, un mécanisme d'adaptation est proposé basé sur la gestion de transactions pour assurer une adaptation dynamique cohérente. Elle assure la terminaison correcte des transactions en cours. Nous proposons pour cela la notion de dépendance transactionnelle : dépendance entre des actions réalisées par des composants différents. Nous proposons la spécification de ces dépendances dans le modèle de variabilité, et de l'exploiter pour décider des fonctions de contrôle dans les composants de l'architecture, des fonctions qui assurent une adaptation cohérente à l'exécution.

Mots clefs : Modélisation de Variabilité, Architecture Logicielle, Adaptation Dynamique Cohérente, Dépendance Transactionnelle, Gestion de Transaction, Ligne de Produit

Abstract

Adaptive software is a class of software which is able to modify its own internal structure and hence its behavior at runtime in response to changes in its operating environment. Adaptive software development has been an emerging research area of software engineering in the last decade. Many existing approaches use techniques issued from software product lines (SPLs) to develop adaptive software architectures. They propose tools, frameworks or languages to build adaptive software architectures but do not guide developers on the process of using them. Moreover, they suppose that all elements in the SPL specified are available in the architecture for adaptation. Therefore, the adaptive software architecture may embed unnecessary elements (components that will never be used) thus limiting the possible deployment targets. On the other hand, the components replacement at runtime remains a complex task since it must ensure the validity of the new version, in addition to preserving the correct completion of ongoing activities.

To cope with these issues, this thesis proposes an adaptive software development process where tasks, roles, and associate artifacts are explicit. The process aims at specifying the necessary information for building adaptive software architectures. The result of such process is an adaptive software architecture that only contains necessary elements for adaptation. On the other hand, an adaptation mechanism is proposed based on transactions management for ensuring consistent dynamic adaptation. Such adaptation must guarantee the system state and ensure the correct completion of ongoing transactions. In particular, transactional dependencies are specified at design time in the variability model. Then, based on such dependencies, components in the architecture include the necessary mechanisms to manage transactions at runtime consistently.

Keywords: Variability Modeling, Software Architecture, Consistent Dynamic Adaptation, Transactional Dependency, Transaction Management, Software Product Line

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

N° d'ordre : 2017IMTA0026

IMT Atlantique Bretagne-Pays de la Loire - www.imt-atlantique.fr

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 03
T +33 (0)2 29 00 11 11
F +33 (0)2 29 00 10 00

Campus de Nantes
4, rue Alfred Kastler - La Chantrerie
CS 20722
44307 Nantes Cedex 3
T +33 (0)2 51 85 81 00
F +33 (0)2 51 85 81 99

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
T +33 (0)2 99 12 70 00
F +33 (0)2 99 12 70 08